Principled Programming

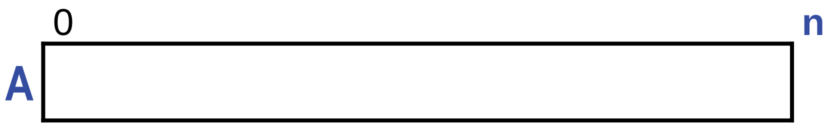
Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Collections

Introduction



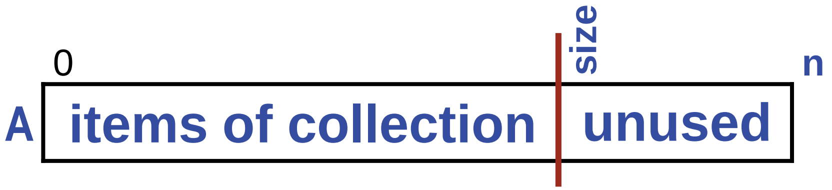
Until now, we have searched in arrays, and have rearranged their values, but the set of values stored in the array has been static, i.e., fixed and unchanging during program execution.

A *collection* is 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

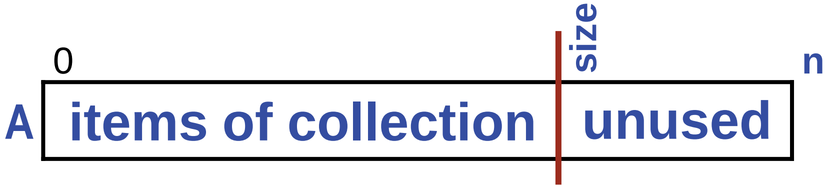
The need to dynamically increase the size of an array leads naturally to the revelation that arrays are *objects*, and to a discussion of Two-Dimensional Arrays.



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

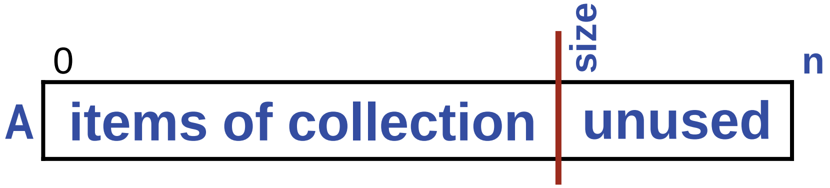
```
/* A[0..size-1] are the current items in A[0..n-1], 0≤size≤n. */
   int A[];   // receptacle for items in a list.
   int size;  // current # of elements in list, 0≤size≤n.
   int n;     // maximum # of elements storable in the list.
```

- 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 **M** and a value *v*, we want (at least) these operations:

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


0

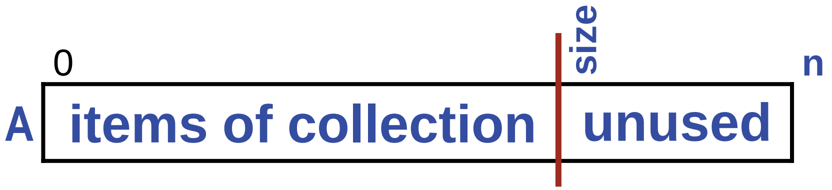
size

n

A items of collection unused

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. */
   A[size] = v;
   size++;
```



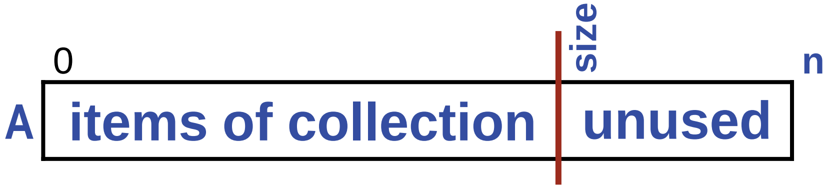
Remove:

```
/* Remove v from A. */
   int k = indexOf(v, A, size);
   if ( k==size ) /* v is not in A. */
   else { size--; A[k] = A[size]; }
```

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

```
/* Return k, a location of v in A, or return size if no v in A. */
static int indexOf(int v; int A[], int size) {
   int k = 0;
   while ( k<size && A[k]!=v ) k++;
   return k;
   }
```

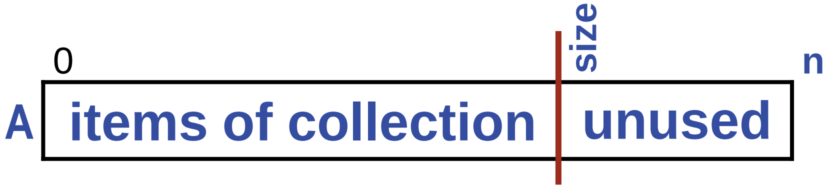
Lists



Membership:

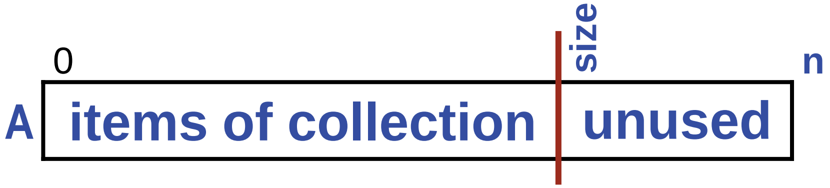
```
/* Set b to true if v is in A, and false otherwise. */
   int k = indexOf(v, A, size);
   boolean b = (k<size);
```

Lists



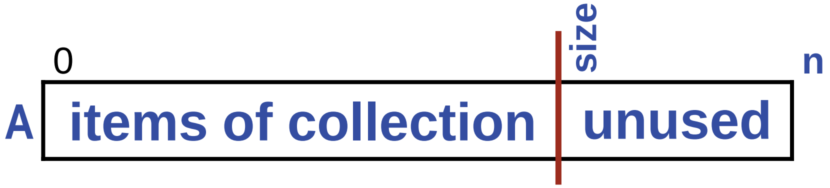
Multiplicity:

```
/* Set m to the multiplicity of v in A. */
   int m = 0;
   for (int k=0; k<size; k++) if ( A[k]==v ) m++;
```



Enumeration:

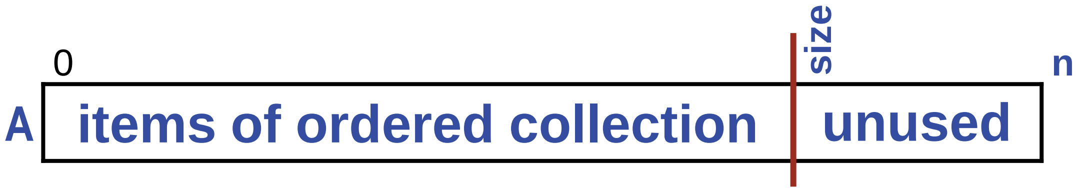
```
/* Enumerate elements of A. */
   for (int k=0; k<size; k++) /* Do whatever for A[k]. */
```



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

Lists
Ordered

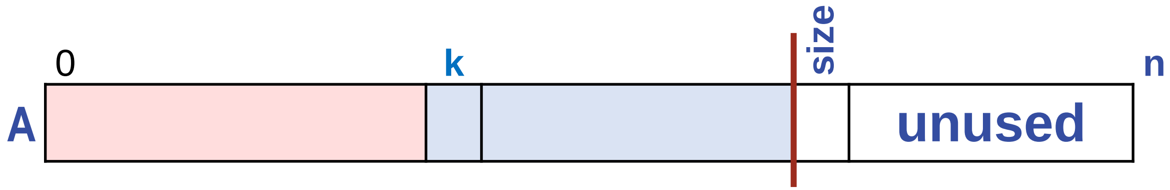


The items in the collection can be maintained as an *indexable* list, sometimes referred to as an *ordered* list.

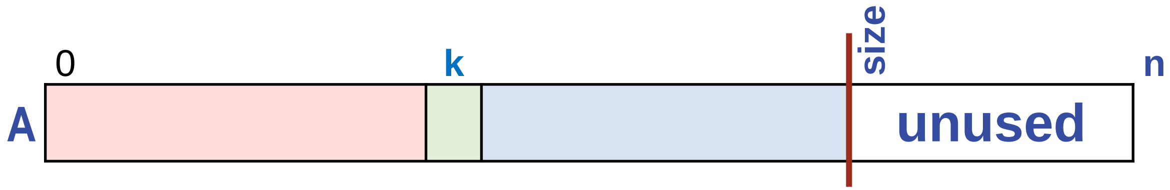
As a consequence, the enumeration of collection members will be in the given order.

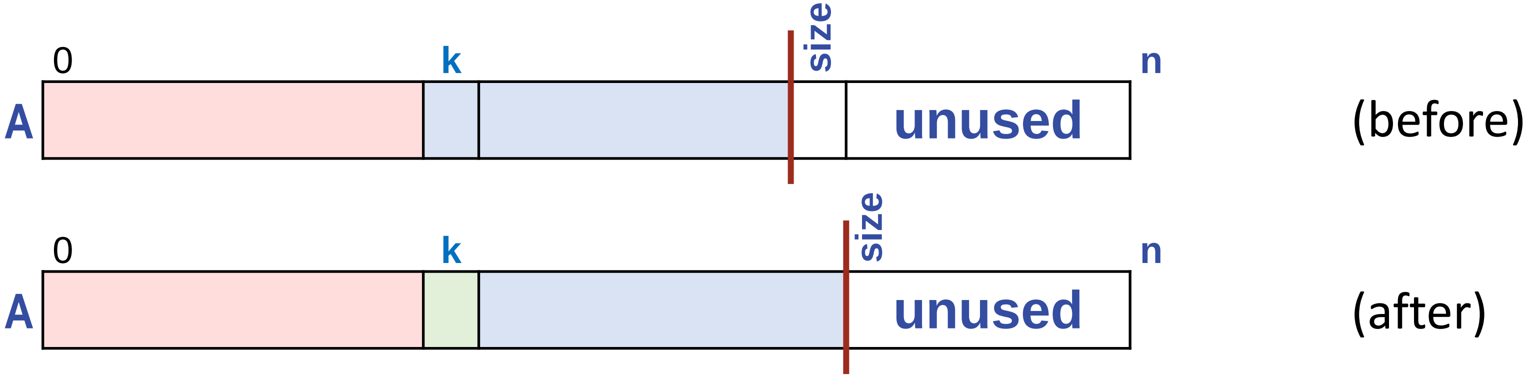
(Note that the ordering is determined by the client, and is not based on the normal arithmetic ordering of values.)

To insert an item (green) at the k*th* index, first shift `A[k..size-1]` right one element:



Conversely, to delete the k*th* item (green), shift `A[k+1..size-1]` left one element:





Add into ordered list:

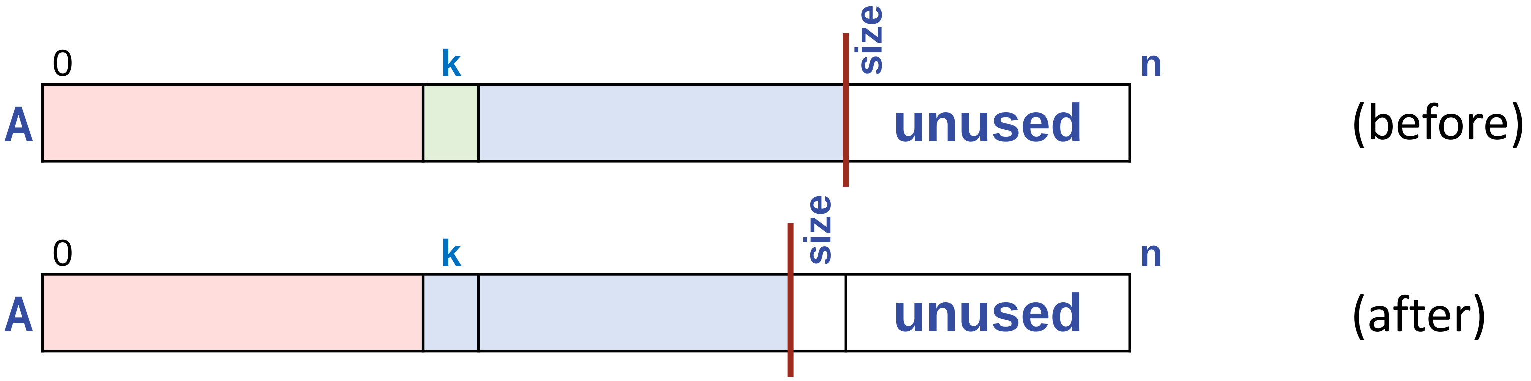
```
/* Add v at position k of A. */
   /* Check index. */
      if (k>size) /* Alert: Bad index. */
   /* Ensure that A has the capacity for another element. */
      if ( size==n ) /* Make room for more values. */
   /* Shift A[k..size-1] right one place to make room for v. */
      for (int j=size-1; j>=k; j--) A[j+1] = A[j];
   A[k] = v;
   size++;
```

Performance:

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

Lists

Ordered



Remove from ordered list:

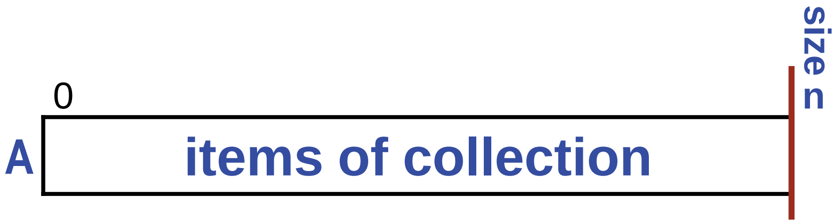
```
/* Remove v from position k of A. */
   /* Check index. */
      if (k>=size) /* Alert: Bad index. */
   size--;
   /* Shift A[k+1..size] left one place to squeeze out v. */
      for (int j=k; j<size; j++) A[j] = A[j+1];
```

Performance:

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

Lists

Unbounded Capacity

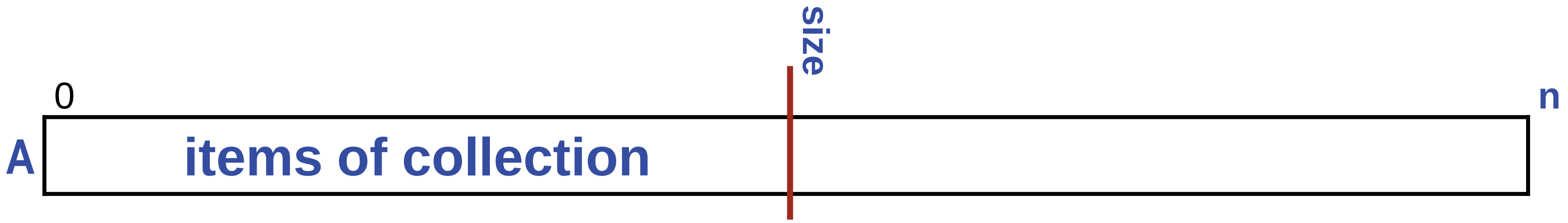


Add: When the array containing items of a collection is full, …

```
/* 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. */
   A[size] = v;
   size++;
```

Lists

Unbounded Capacity



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 = ensureCapacity(A); n = A.length; }
   A[size] = v;
   size++;
```

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

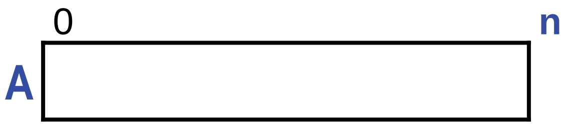
This statement updates n, where we have maintained the length of an array A in a separate variable. We have done so, but didn’t need to because it was available as `A.length` all along.

Lists

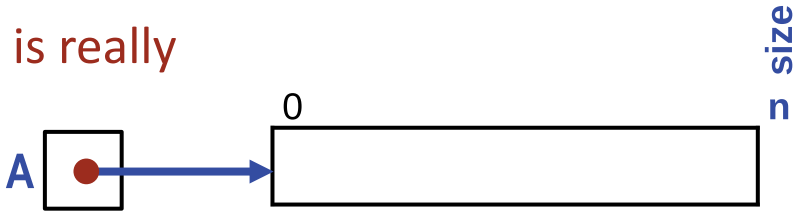
Unbounded Capacity

Add: To understand how this works, know that

What has been depicted as



is really



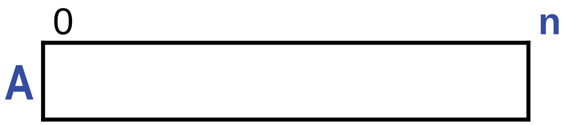
The value in A (the ●) is called a *reference* to an *object* (the **`int`** array).

Lists

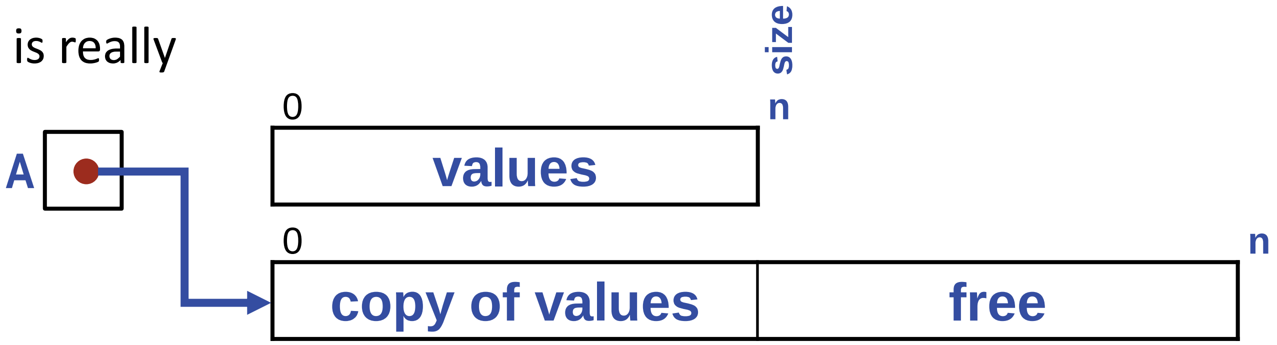
Unbounded Capacity

Add: To understand how this works, know that

What has been depicted as



is really



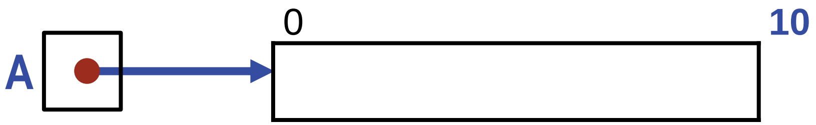
The value in A (the ●) is called a *reference* to an *object* (the **`int`** array).

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:

```
int A[] = new int[10];
int B[] = A;
A[0] = 7;
B[0] = 8;
System.out.println( 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.



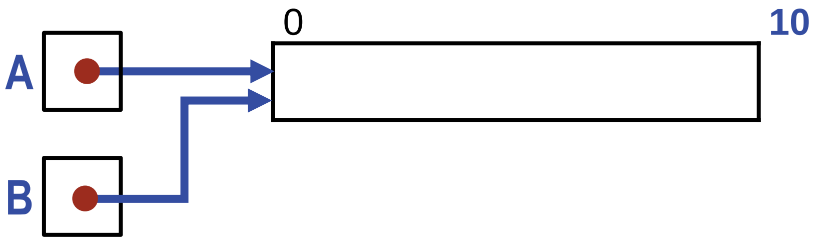
Lists

Unbounded Capacity

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

```
int A[] = new int[10];
int B[] = A;
A[0] = 7;
B[0] = 8;
System.out.println( A[0] ); // What does this line print?
```

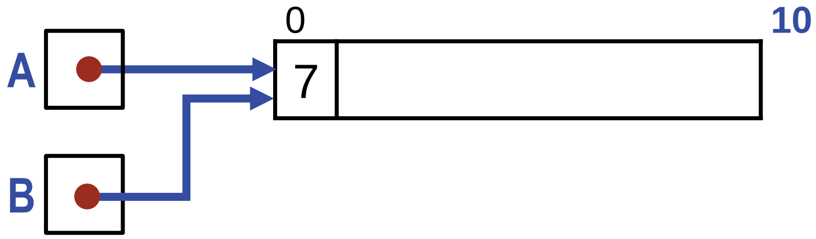
The second line declares B, and assigns the contents of A (the reference) to B.



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

```
int A[] = new int[10];
int B[] = A;
A[0] = 7;
B[0] = 8;
System.out.println( A[0] ); // What does this line print?
```

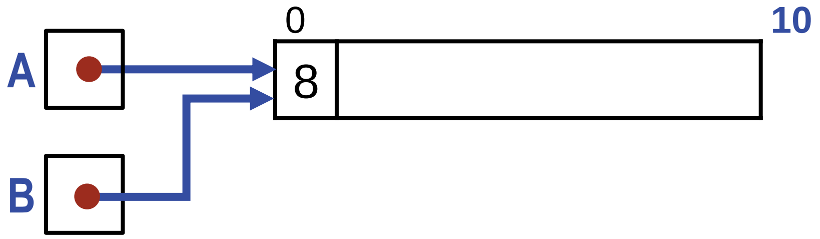
The third line assigns 7 to be the contents of `A[0]`, the 0th variable in the array object referred to by A.



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

```
int A[] = new int[10];
int B[] = A;
A[0] = 7;
B[0] = 8;
System.out.println( 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.



Lists

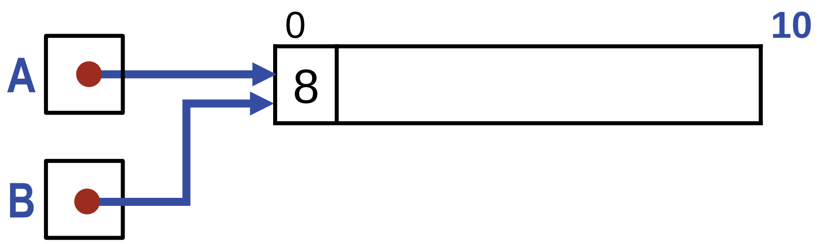
Unbounded Capacity

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

```
int A[] = new int[10];
int B[] = A;
A[0] = 7;
B[0] = 8;
System.out.println( 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.

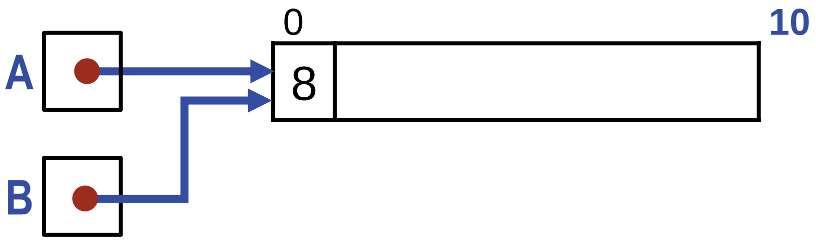
It prints 8.



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

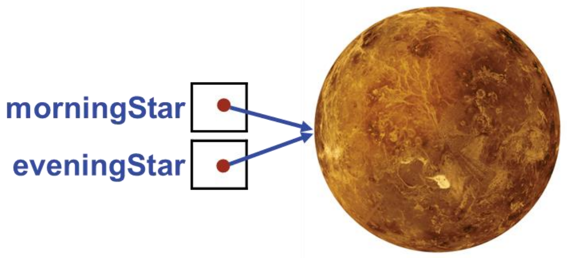
```
int A[] = new int[10];
int B[] = A;
A[0] = 7;
B[0] = 8;
System.out.println( A[0] ); // What does this line print?
```

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



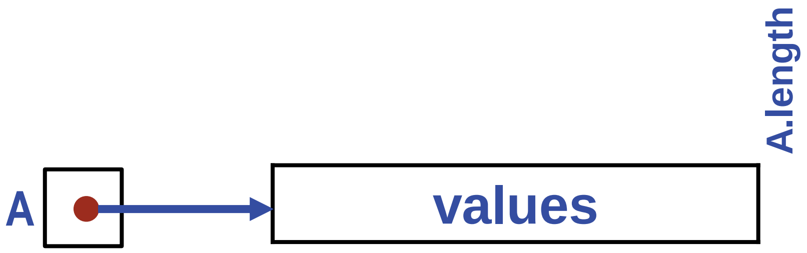
It prints 8.

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.



Unbounded Capacity

EnsureCapacity:

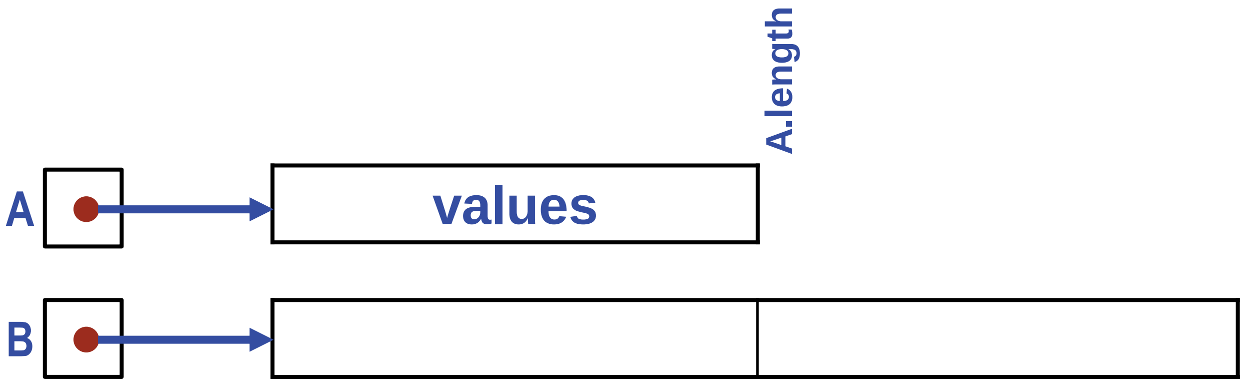


```
/* Return a reference to a copy of A in an object that is twice as long. */
static int[] ensureCapacity( int A[] ) {
 /* Make B refer to an object that is twice as long as A. */
    int B[] = new int[2*A.length];
 /* Copy the values from A (the old object) to B (the new object). */
    for (int k=0; k<A.length; k++) B[k] = A[k];
 return B;
 } /* ensureCapacity */
```

Lists

Unbounded Capacity

EnsureCapacity:

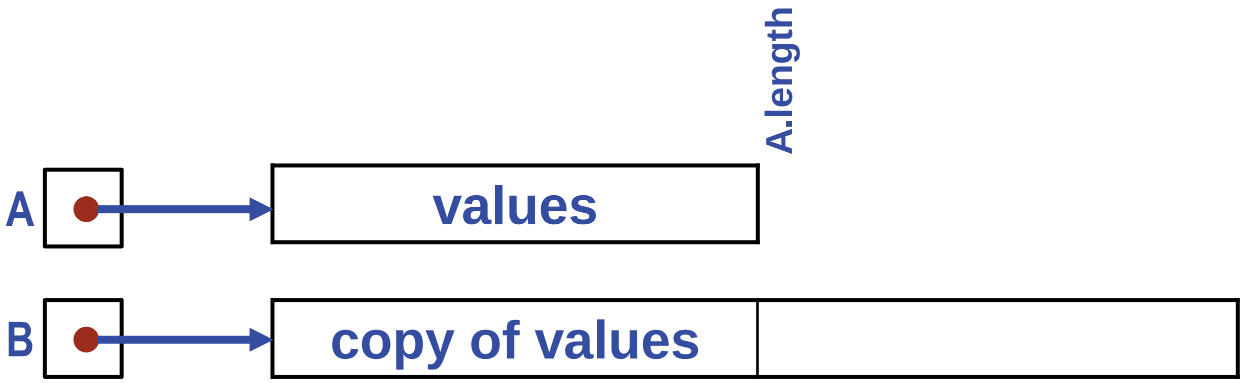


```
/* Return a reference to a copy of A in an object that is twice as long. */
static int[] ensureCapacity( int A[] ) {
 /* Make B refer to an object that is twice as long as A. */
    int B[] = new int[2*A.length];
 /* Copy the values from A (the old object) to B (the new object). */
    for (int k=0; k<A.length; k++) B[k] = A[k];
 return B;
 } /* ensureCapacity */
```

Lists

Unbounded Capacity

EnsureCapacity:

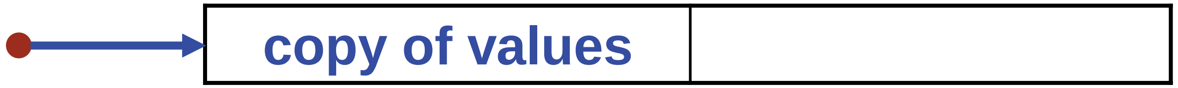


```
/* Return a reference to a copy of A in an object that is twice as long. */
static int[] ensureCapacity( int A[] ) {
 /* Make B refer to an object that is twice as long as A. */
    int B[] = new int[2*A.length];
 /* Copy the values from A (the old object) to B (the new object). */
    for (int k=0; k<A.length; k++) B[k] = A[k];
 return B;
 } /* ensureCapacity */
```

Lists

Unbounded Capacity

EnsureCapacity:



```
/* Return a reference to a copy of A in an object that is twice as long. */
static int[] ensureCapacity( int A[] ) {
 /* Make B refer to an object that is twice as long as A. */
    int B[] = new int[2*A.length];
 /* Copy the values from A (the old object) to B (the new object). */
    for (int k=0; k<A.length; k++) B[k] = A[k];
 return B;
 } /* ensureCapacity */
```

Array Parameters

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

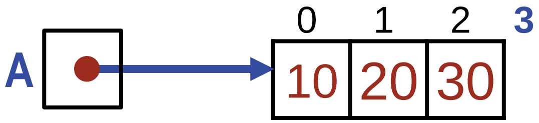
```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

Lists

Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

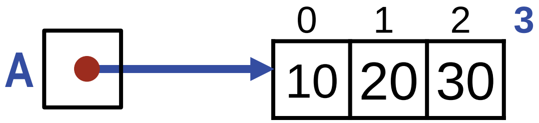


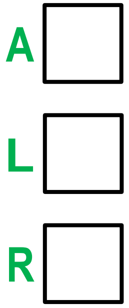
Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

```
static void Reverse( int A[], int L, int R ) {
   while ( L<R ) {
      /* Swap A[L] and A[R]. */
         int temp = A[L]; A[L]=A[R]; A[R]=temp;
      L++; R--;
      }
   }
```





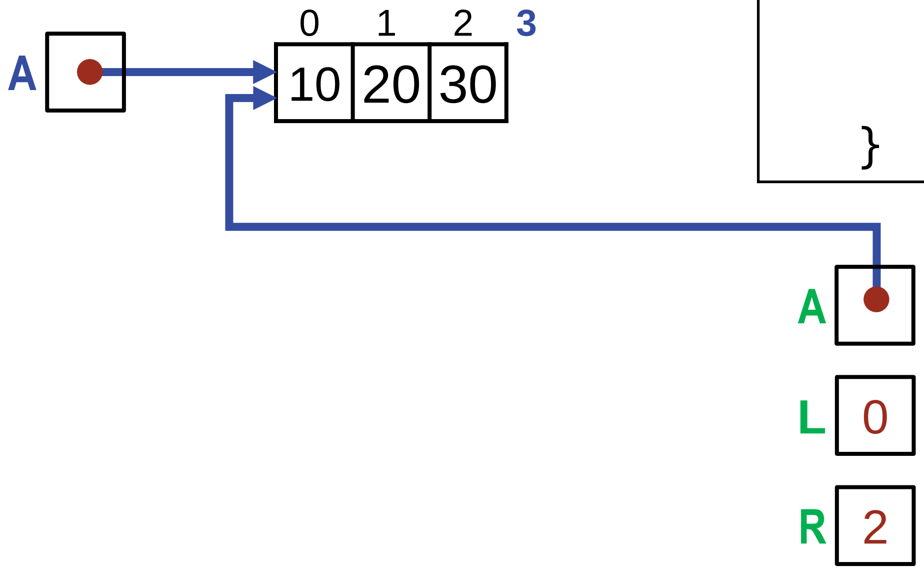
Lists

Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

```
static void Reverse( int A[], int L, int R ) {
   while ( L<R ) {
      /* Swap A[L] and A[R]. */
         int temp = A[L]; A[L]=A[R]; A[R]=temp;
      L++; R--;
      }
   }
```



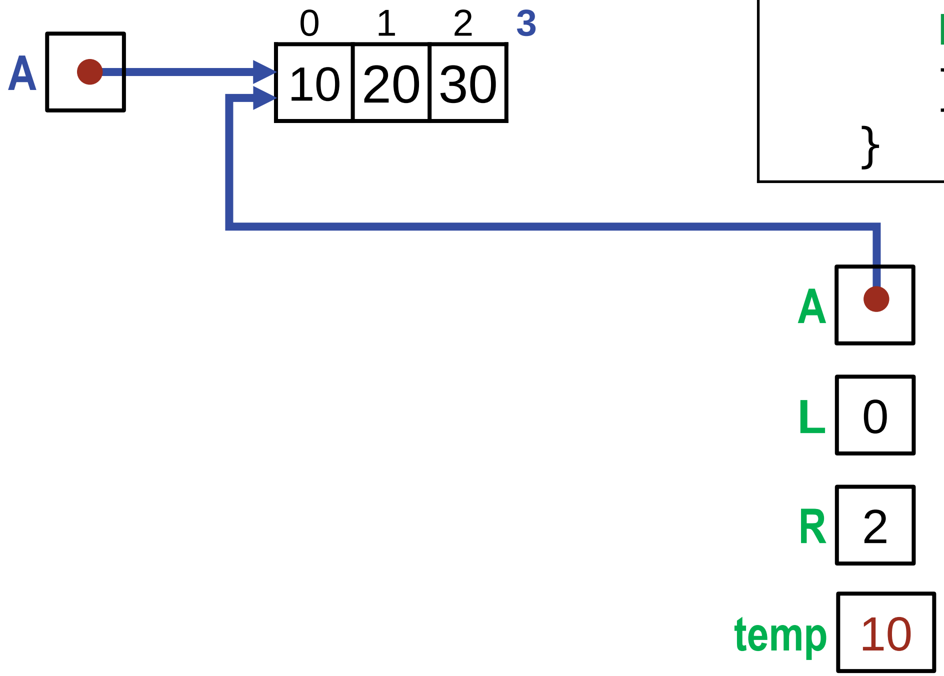
The blue A and the green A are aliases.

Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

```
static void Reverse( int A[], int L, int R ) {
  while ( L<R ) {
    /* Swap A[L] and A[R]. */
      int temp = A[L]; A[L]=A[R]; A[R]=temp;
    L++; R--;
    }
  }
```



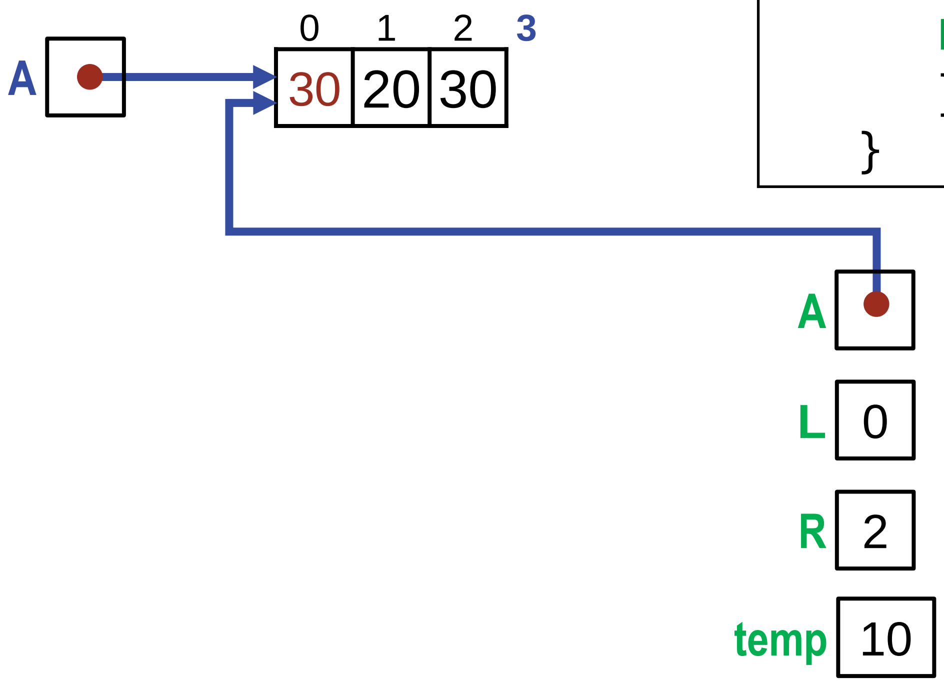
Lists

Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

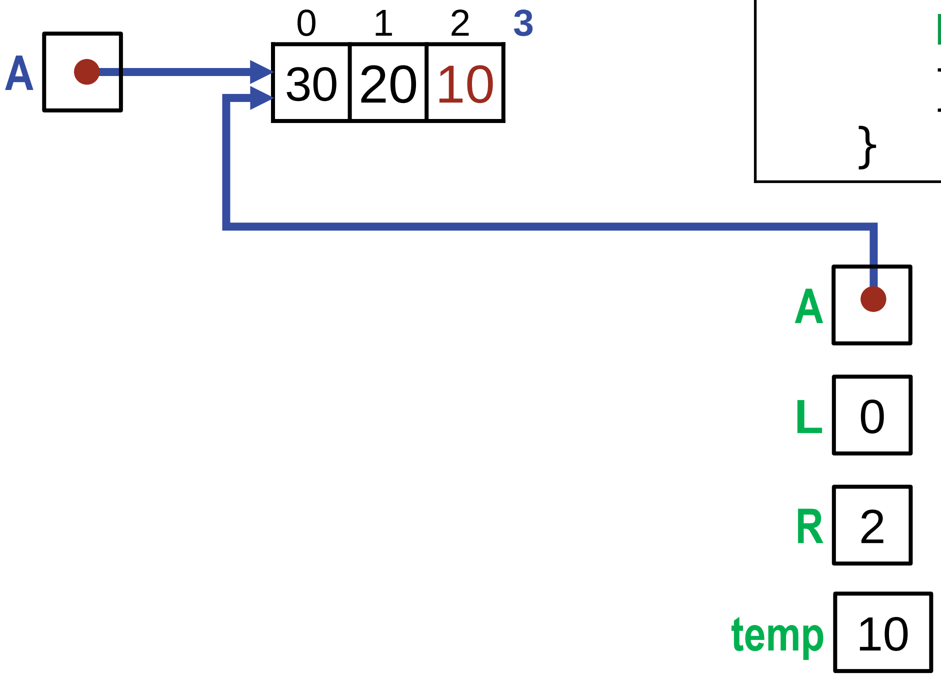
```
static void Reverse( int A[], int L, int R ) {
   while ( L<R ) {
      /* Swap A[L] and A[R]. */
         int temp = A[L]; A[L]=A[R]; A[R]=temp;
      L++; R--;
      }
   }
```



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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

```
static void Reverse( int A[], int L, int R ) {
   while ( L<R ) {
      /* Swap A[L] and A[R]. */
         int temp = A[L]; A[L]=A[R]; A[R]=temp;
      L++; R--;
      }
   }
```

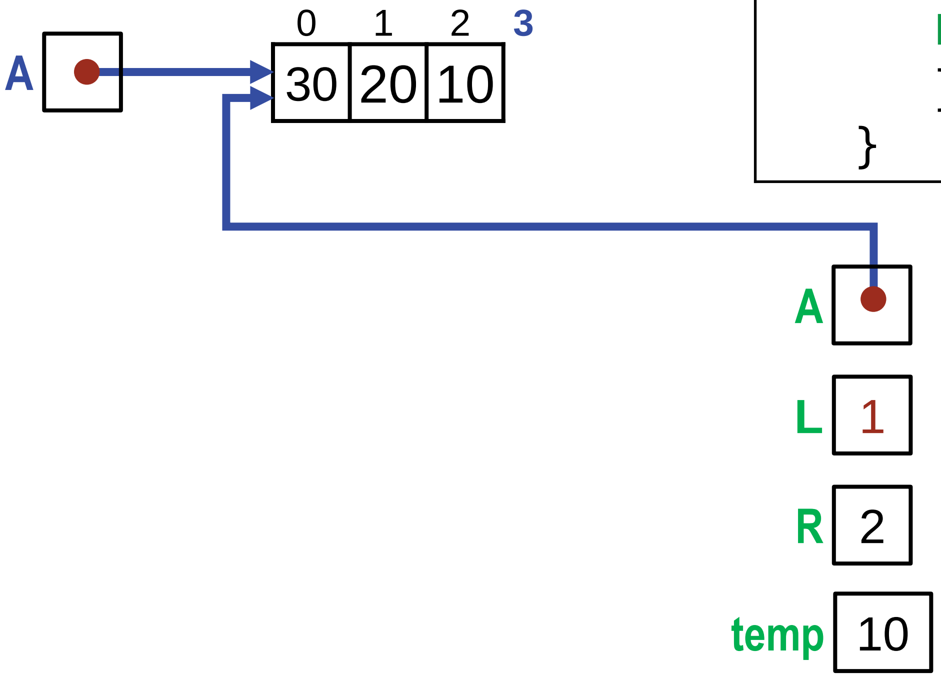


Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

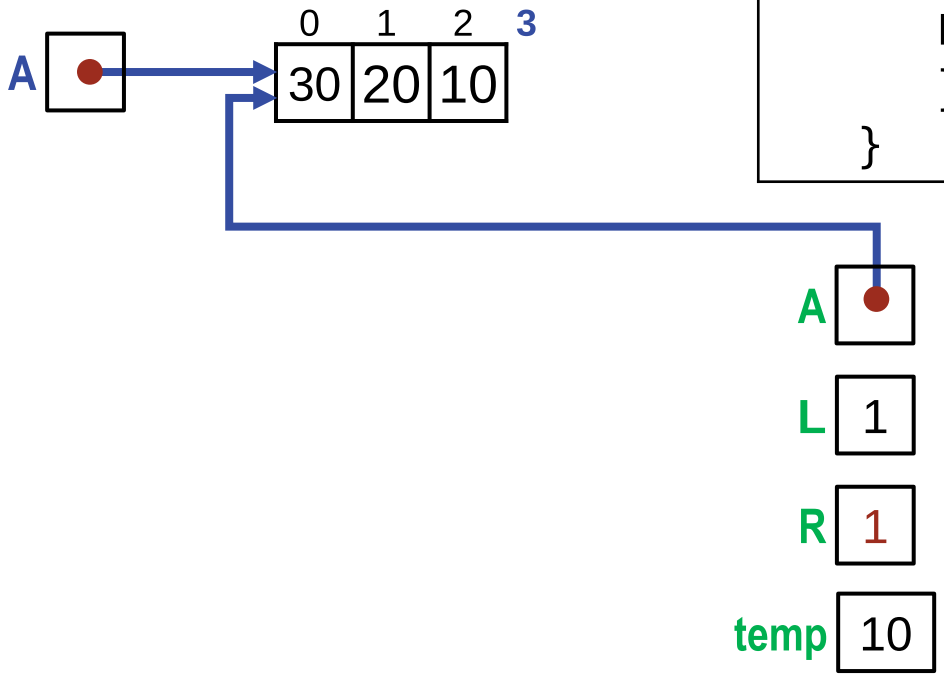
```
static void Reverse( int A[], int L, int R ) {
   while ( L<R ) {
      /* Swap A[L] and A[R]. */
         int temp = A[L]; A[L]=A[R]; A[R]=temp;
      L++; R--;
      }
   }
```



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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

```
static void Reverse( int A[], int L, int R ) {
   while ( L<R ) {
      /* Swap A[L] and A[R]. */
         int temp = A[L]; A[L]=A[R]; A[R]=temp;
      L++; R--;
      }
   }
```

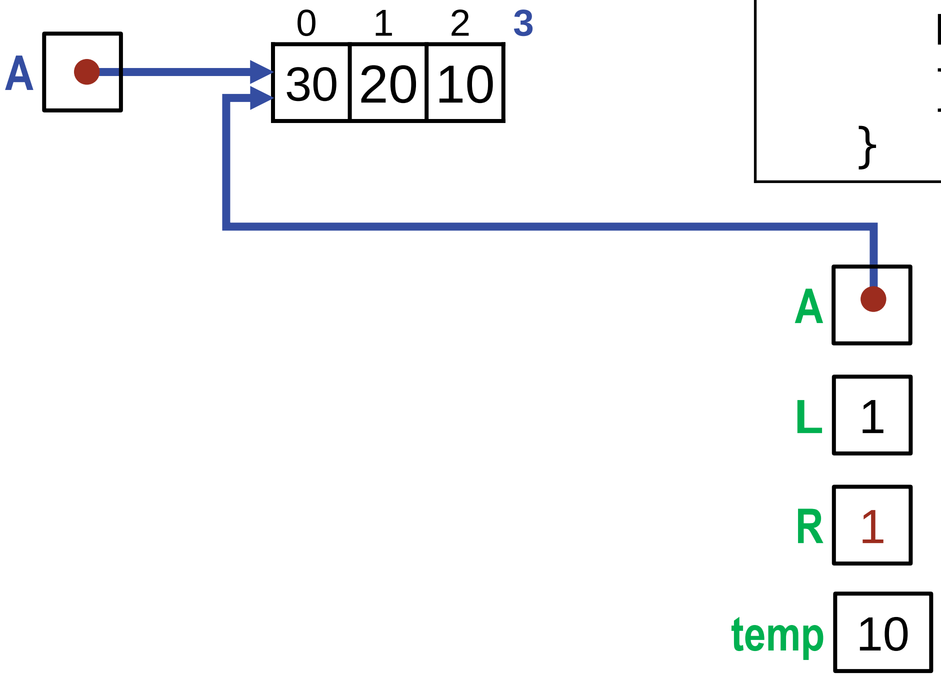


Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```

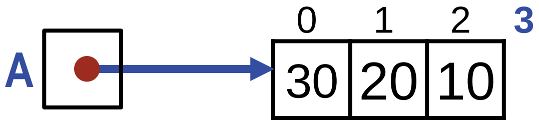
```
static void Reverse( int A[], int L, int R ) {
    while ( L<R ) {
        /* Swap A[L] and A[R]. */
            int temp = A[L]; A[L]=A[R]; A[R]=temp;
        L++; R--;
        }
    }
```



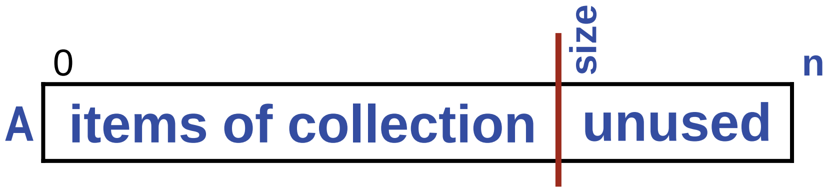
Array Parameters

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

```
int A[] = { 10, 20, 30 };
Reverse(A,0,2);
```



Critique



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:

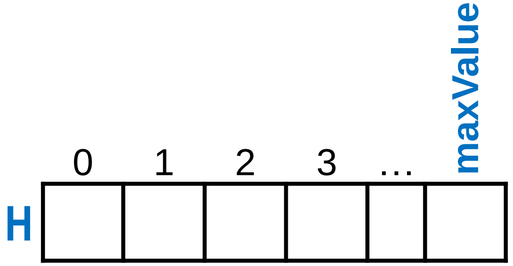
- The collection has no single name, and thus it is not easily manipulated as one thing.
- The collection’s implementation details are not hidden, and thus your program can both break the data structure’s representation invariant and come to excessively depend on its details.

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.

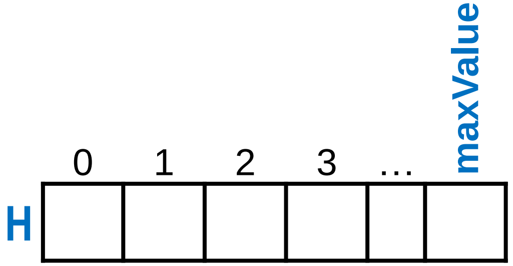
Histograms



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

```
/* Collection of items in range 0..maxValue, where multiplicity of v is H[v]. */
   int H[0..maxValue];
```

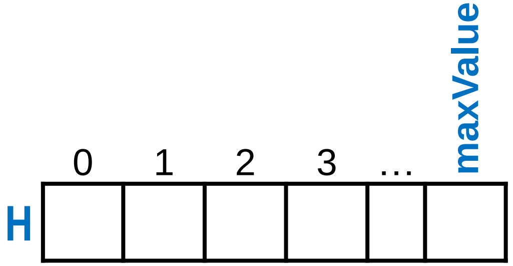
Histograms



Add:

```
/* Add v to H. */
   H[v]++;
```

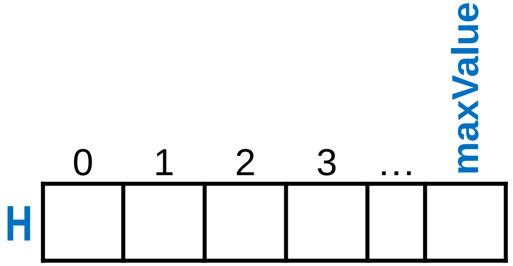
Histograms



Remove:

```
/* Remove v from H. */
   if ( H[k]==0 ) /* Alarm: attempt to remove a value not in H. */
   else H[k]--;
```

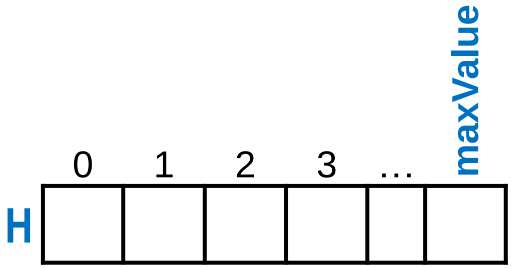
Histograms



Membership:

```
/* b = true iff v is in H. */
   boolean b = (H[v]>0);
```

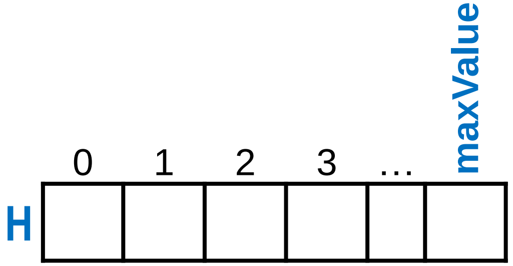
Histograms



Multiplicity:

```
/* m = Multiplicity of v in H. */
   int m = H[v];
```

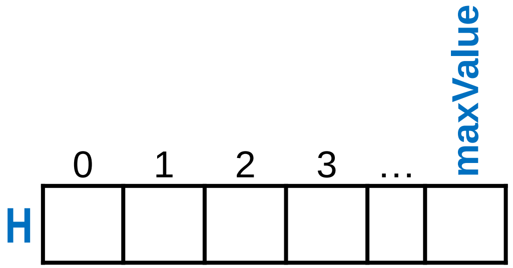
Histograms



Enumeration:

```
/* Enumerate elements of H. */
   for (int k=0; k<=maxValue; k++)
      for (int j=1; j<=H[k]; j++)
         /* Enumerate k. */
```

Histograms

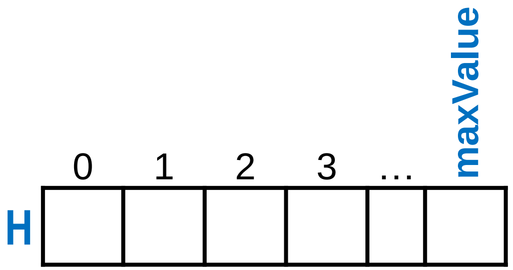


Performance:

Operation	Steps
add	constant
remove	constant
membership	constant
multiplicity	constant
enumeration	linear in `maxValue` + number of elements in the multiset

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

Histograms



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..maxValue]`.
- Associated values. The histogram representation does not provide an obvious way to represent the associated value components of ⟨key,value⟩ pairs. In contrast, to represent a multiset of ⟨key,value⟩ 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 ⟨key,value⟩-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 (int r=1; r<13; r++)
         for (int c=0; c<r; c++)
            /* 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.

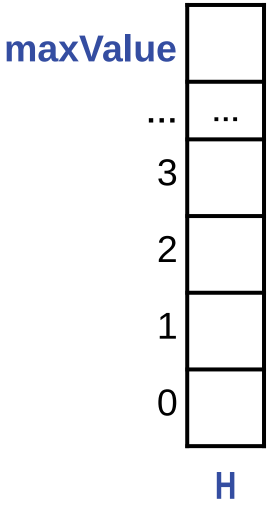
```
int N = 12*12*12+11*11*11+1; // (Max r)^3+(max c)^3+1, for r!=c in [0..12].
int H[] = new int[N];        // H[k] = # of {r,c}, r!=c, s.t. k=r^3+c^3.
/* Confirm Ramanujan’s claim that 1729 is the smallest number that is the
   sum of two positive cubes in two different ways. */
   /* Let H be a histogram of r^3+c^3, for each set {r,c} of distinct
      nonnegative integers that are no larger than 12. */
      for (int r=1; r<13; r++)
         for (int c=0; c<r; c++)
            H[r*r*r+c*c*c]++;
    /* Output non-zero bins of histogram H. */
       for (int k=0; k<N; k++)
          if ( H[k]>0 ) System.out.println(k + " " + H[k]);
```

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

```
int N = 12*12*12+11*11*11+1; // (Max r)^3+(max c)^3+1, for r!=c in [0..12].
int H[] = new int[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 (int r=1; r<13; r++)
         for (int c=0; c<r; c++)
            H[r*r*r+c*c*c]++;
   /* Let k be smallest index s.t. H[k]>1. */
      int k=0;
      while ( H[k]<2 ) k++;
   if ( H[k]==2 && k==1729 ) System.out.println( "confirmed" );
   else System.out.println( "not confirmed" );
```

Hash Tables

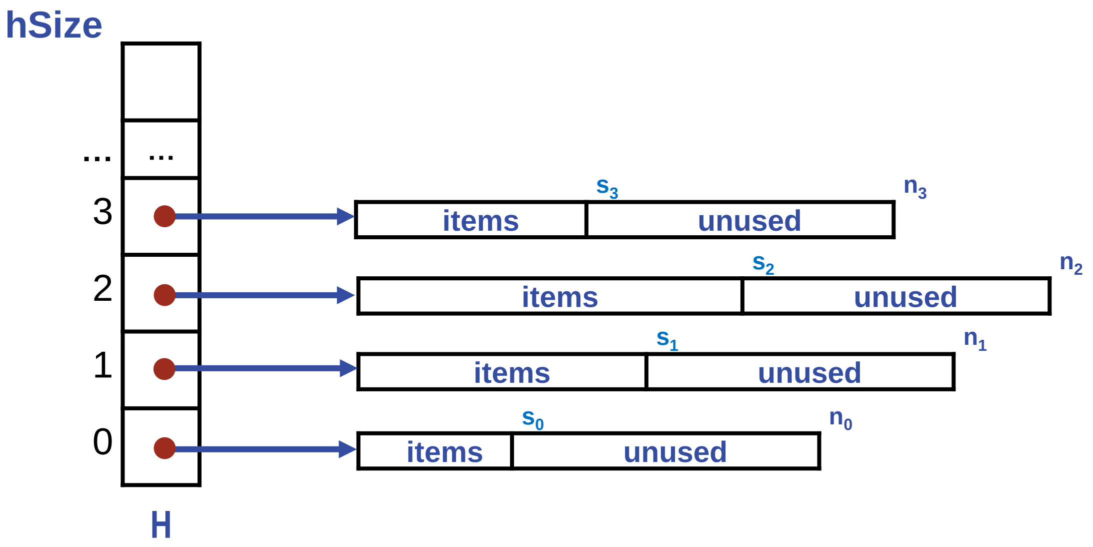
Hash Tables combine the good aspects of lists and histograms.



Start with a histogram `H[0..maxValues]`.

Hash Tables

Hash Tables combine the good aspects of lists and histograms.

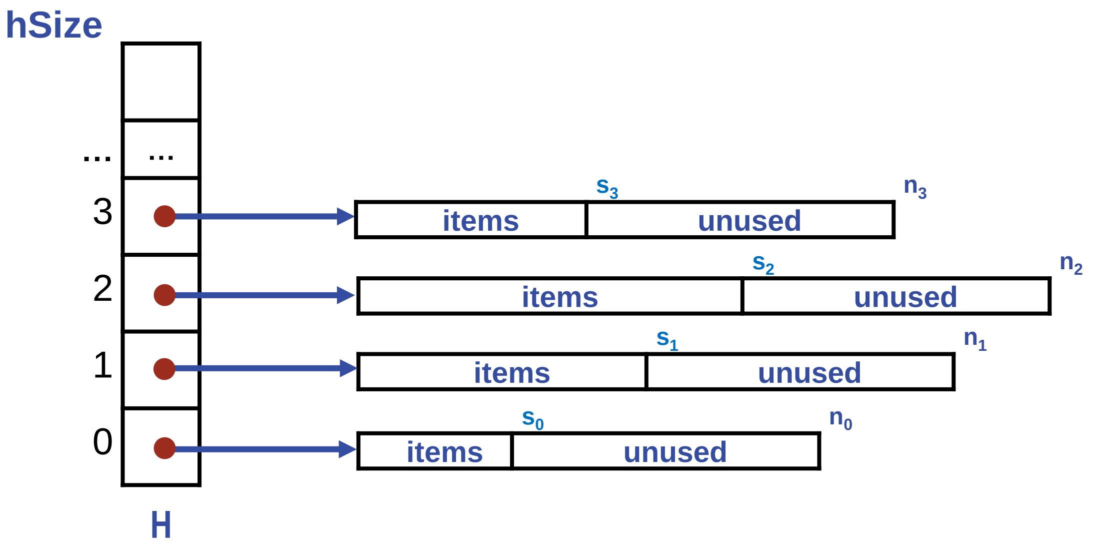


To implement a collection of items that have keys of an arbitrary type *t*:

- Introduce function hash: *t* ⟶ 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 **mod** `hSize]`.
- Dynamically adjust `hSize`, as needed, to keep H and sub-collections not too big and not too small.

Hash Tables

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:

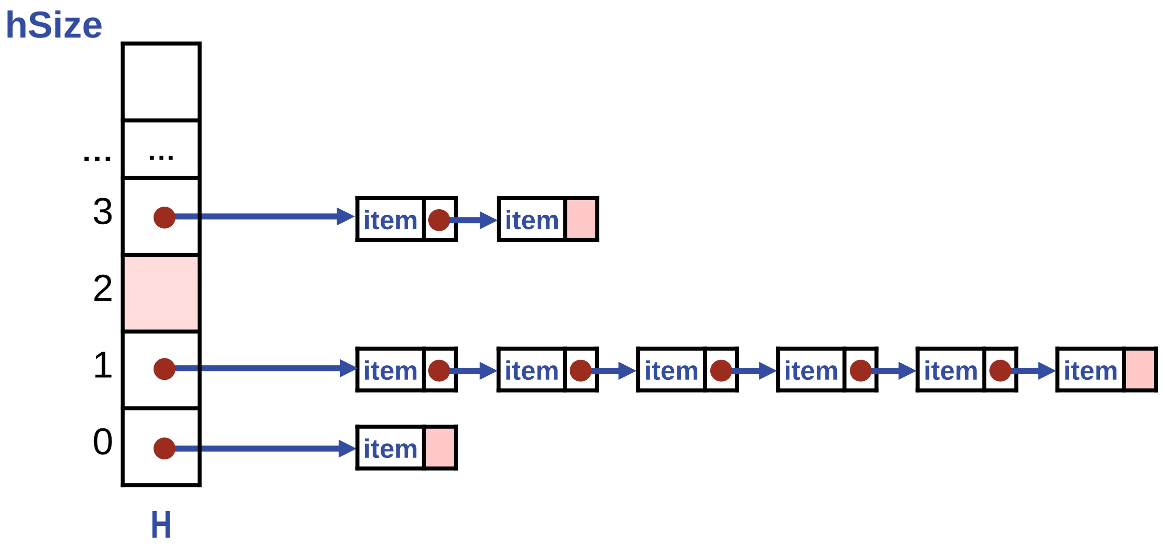
- Doubling `hSize` and reallocating items to appropriate new half-length sub-collection.

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

- Introduce function hash: *t* → 0..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 **mod** `hSize`].
- Dynamically adjust `hSize`, as needed, to keep H and sub-collections not too big and not too small.

Hash Tables

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*.

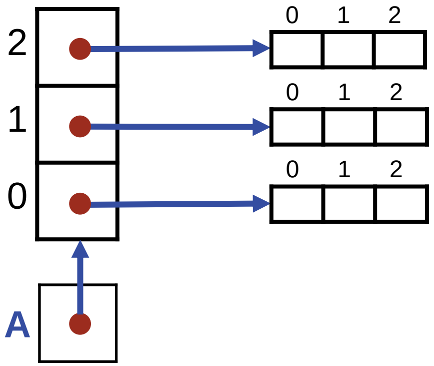
To implement a collection of items that have keys of an arbitrary type t:

- Introduce function 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 **mod** `hSize]`.
- Dynamically adjust `hSize`, as needed, to keep H and sub-collections not too big and not too small.

Two-Dimensional Arrays

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

```
int A[][] = new int[3][3];
```



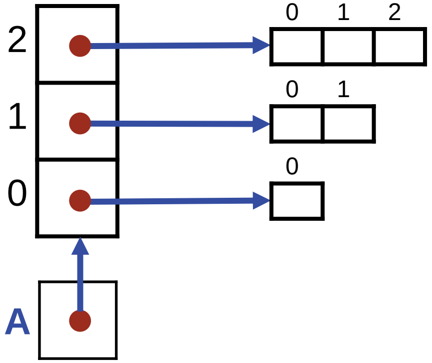
Two-Dimensional Arrays

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. */
   int A[][] = new int[3][];
   for (int r=0; r<3; r++) A[r] = new int[r+1];
```

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



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