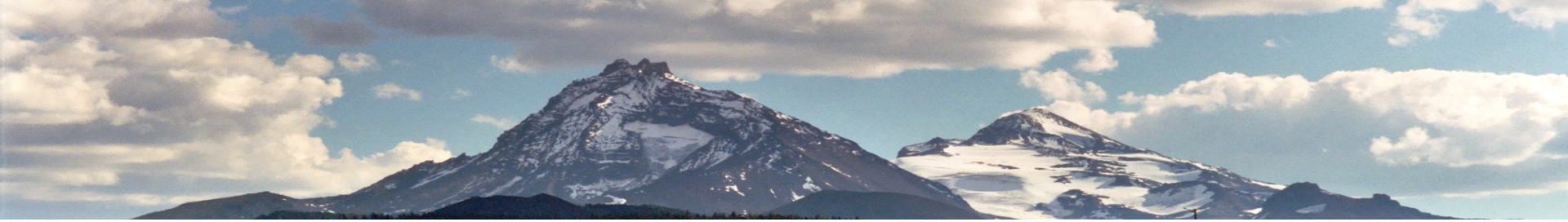


Data Reorganization Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science

UNIVERSITY OF OREGON

Table of Contents

Gather: Serial Implementation

```
1  template<typename Data, typename Idx>
2  void gather(
3      size_t n, // number of elements in data  collection
4      size_t m, // number of elements in index  collection
5      Data a[], // input data collection  (n  elements)
6      Data A[], // output data collection  (m elements)
7      Idx idx[] // input index collection  (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check array bounds
12         A[i] = a[j]; // perform random read
13     }
14 }
```

Serial implementation of gather in pseudocode

Gather: Serial Implementation

```
1  template<typename Data, typename Idx>
2  void gather(
3      size_t n, // number of elements in data  collection
4      size_t m, // number of elements in index  collection
5      Data a[], // input data collection  (n  elements)
6      Data A[], // output data collection  (m elements)
7      Idx idx[] // input index collection  (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check array bounds
12         A[i] = a[j]; // perform random read
13     }
14 }
```

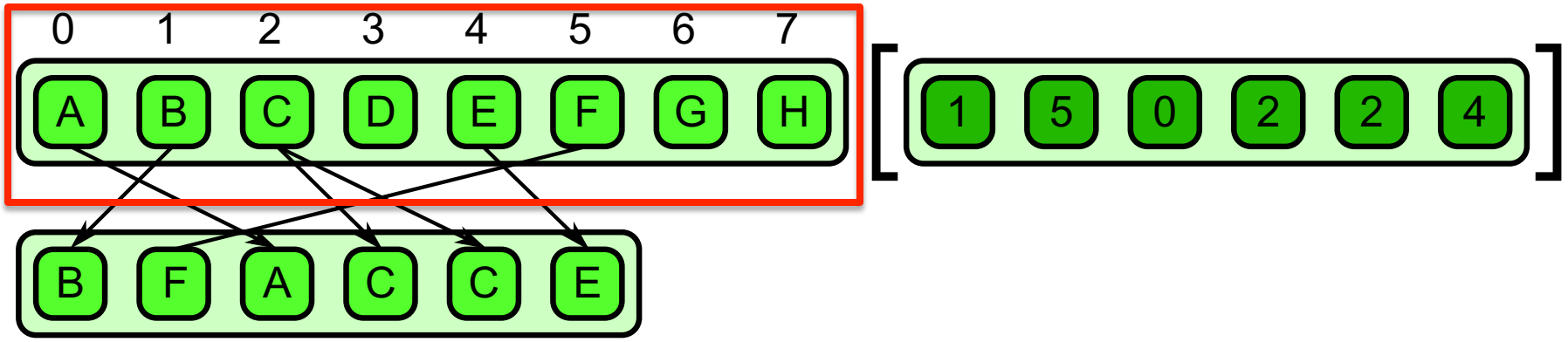
Parallelize over for loop to perform random read

Serial implementation of gather in pseudocode

Gather: Defined

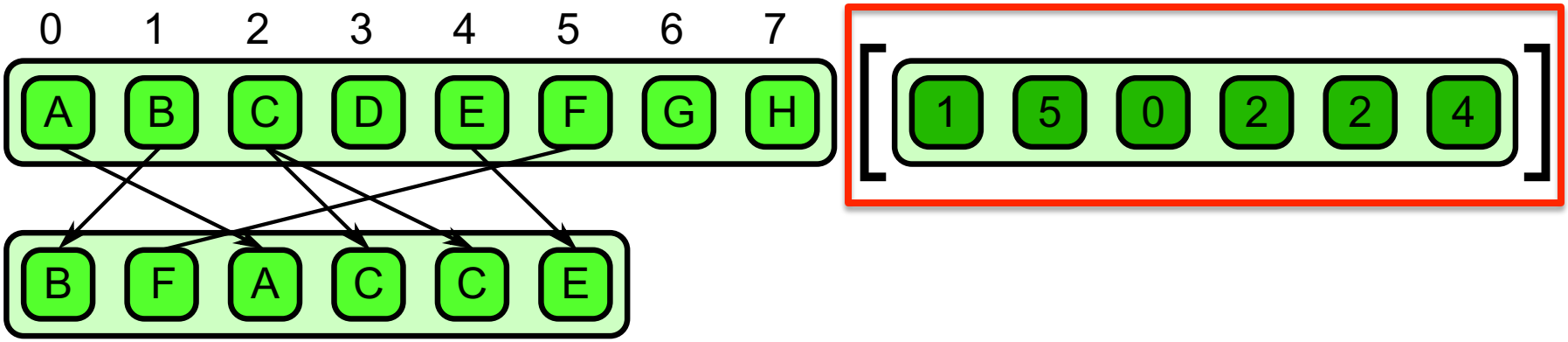
- Results from the combination of a map with a random read

Gather: Defined



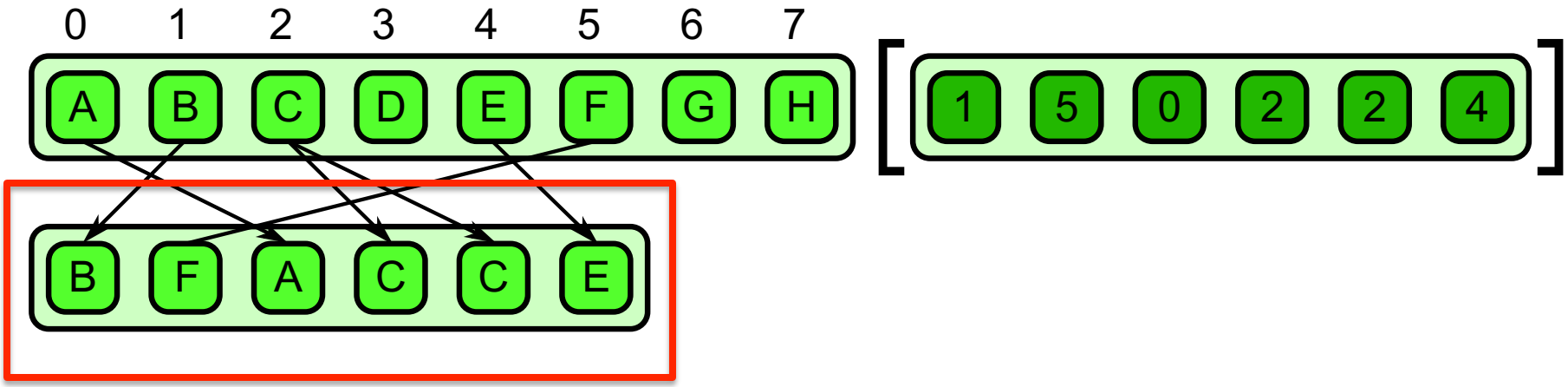
Given a **collection of locations** (address or array indices)

Gather: Defined



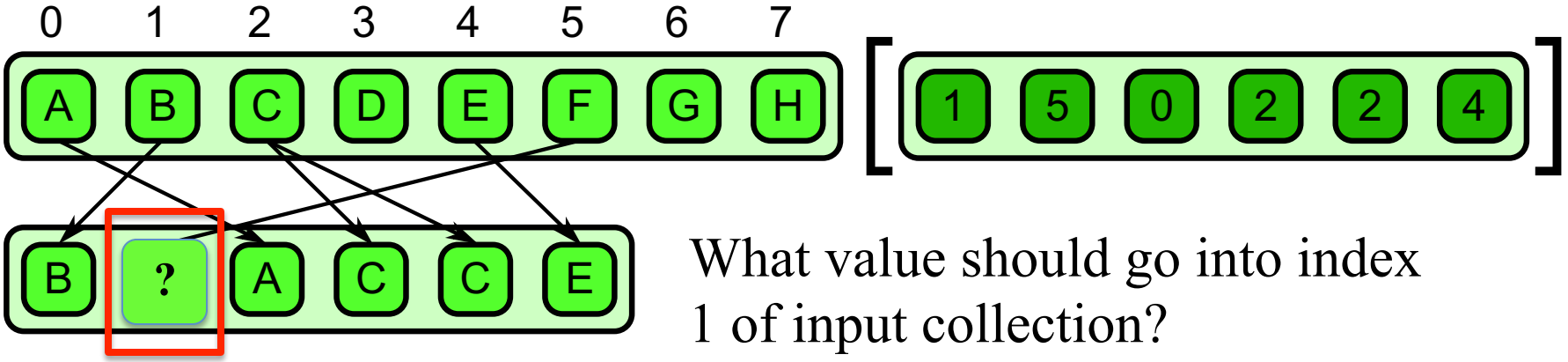
Given a collection of locations (address or array indices) and a **source array**

Gather: Defined



Given a collection of locations (address or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an **output collection**

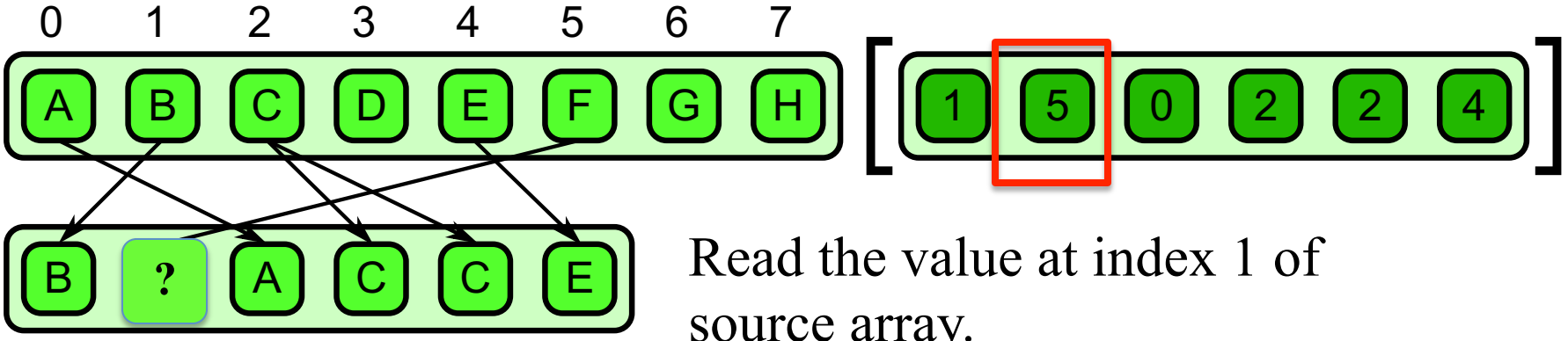
Gather: Defined



What value should go into index 1 of input collection?

Given a collection of locations (address or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an **output collection**

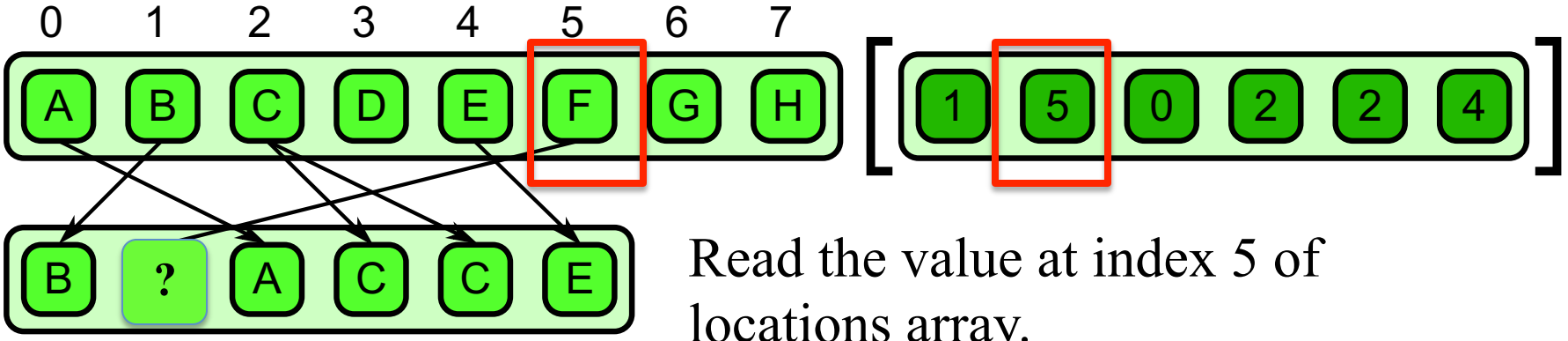
Gather: Defined



Read the value at index 1 of source array.

Given a collection of locations (address or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an **output collection**

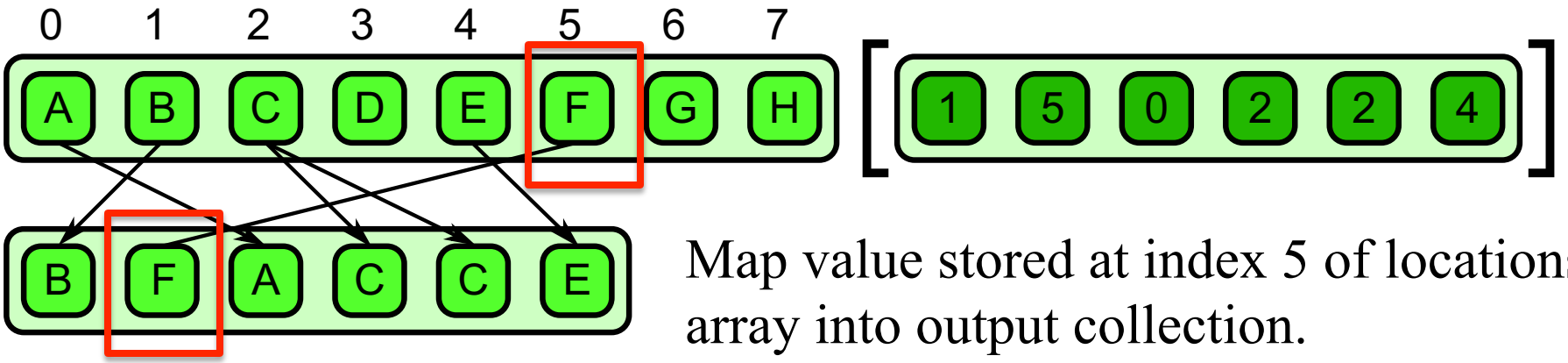
Gather: Defined



Read the value at index 5 of locations array.

Given a collection of locations (address or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined

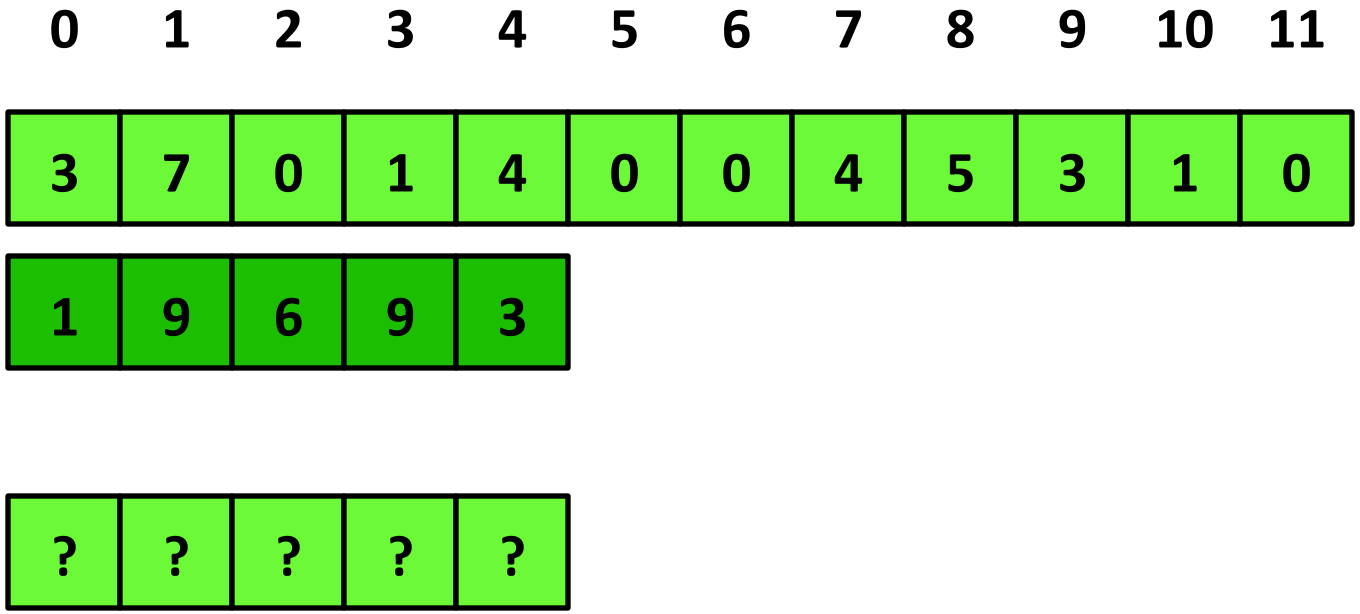


Map value stored at index 5 of locations array into output collection.

Given a collection of locations (address or array indices) and a source array, gather collects all the data from the source array at the given locations and places them into an **output collection**

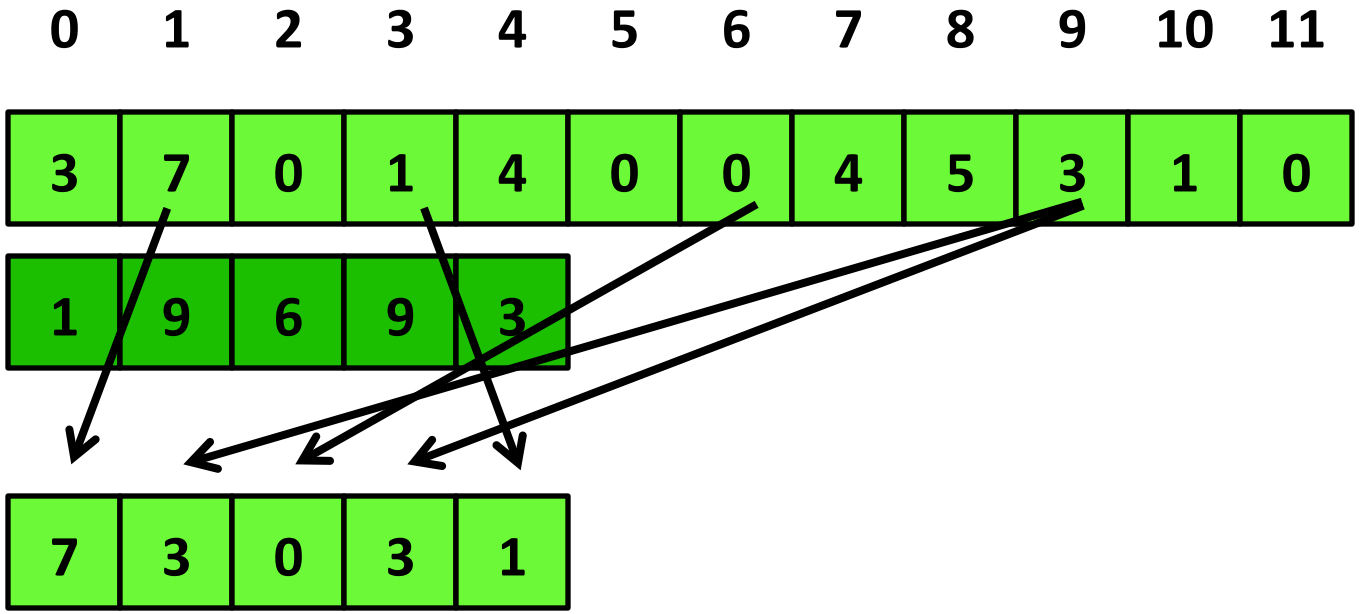
Quiz 1

Given the following locations and source array, use a gather to determine what values should go into the output collection:

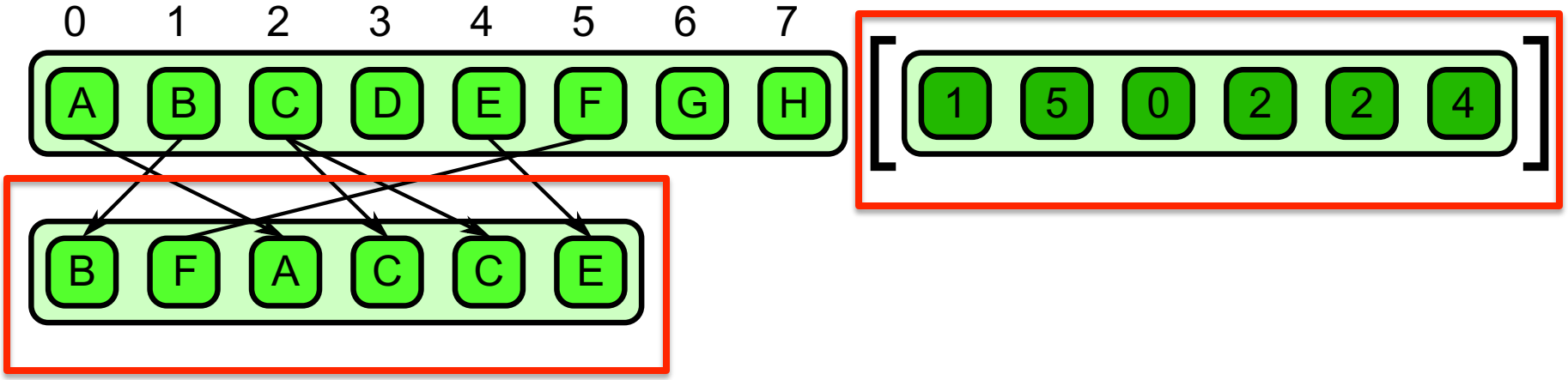


Quiz 1

Given the following locations and source array, use a gather to determine what values should go into the output collection:

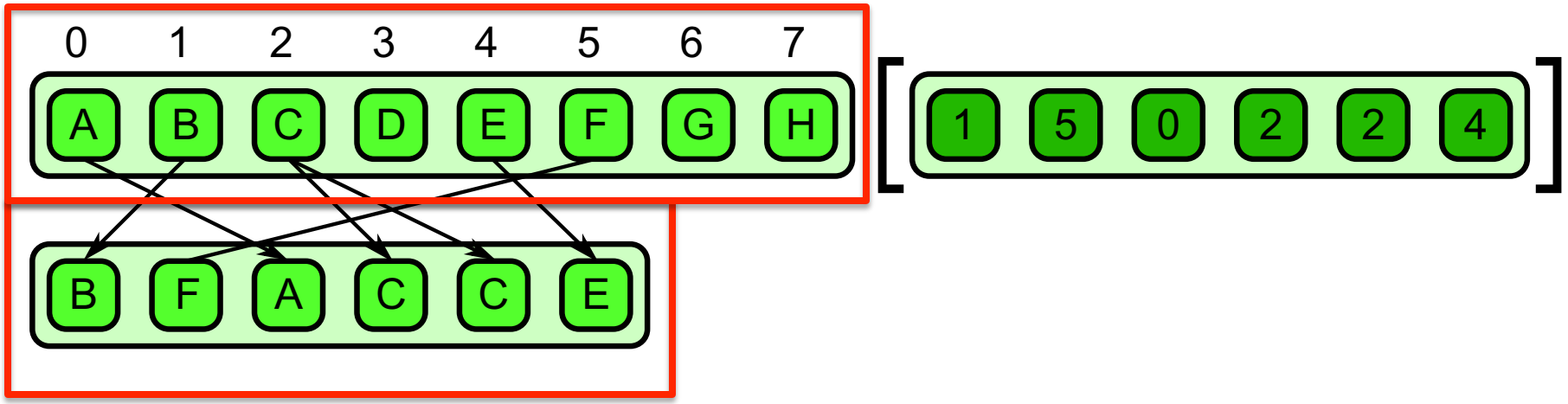


Gather: Array Size



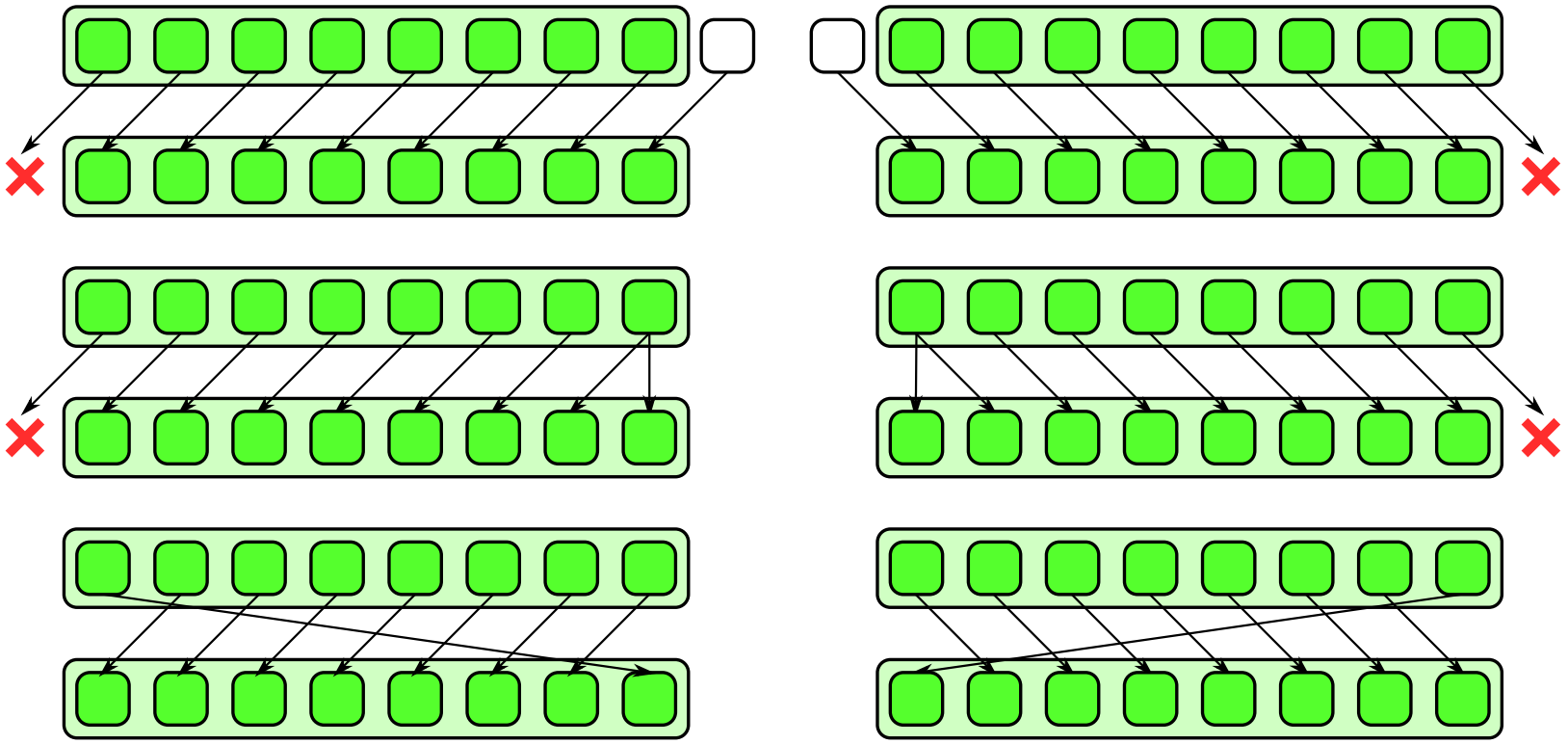
- Output data collection has the same number of elements as the number of indices in the index collection

Gather: Array Size



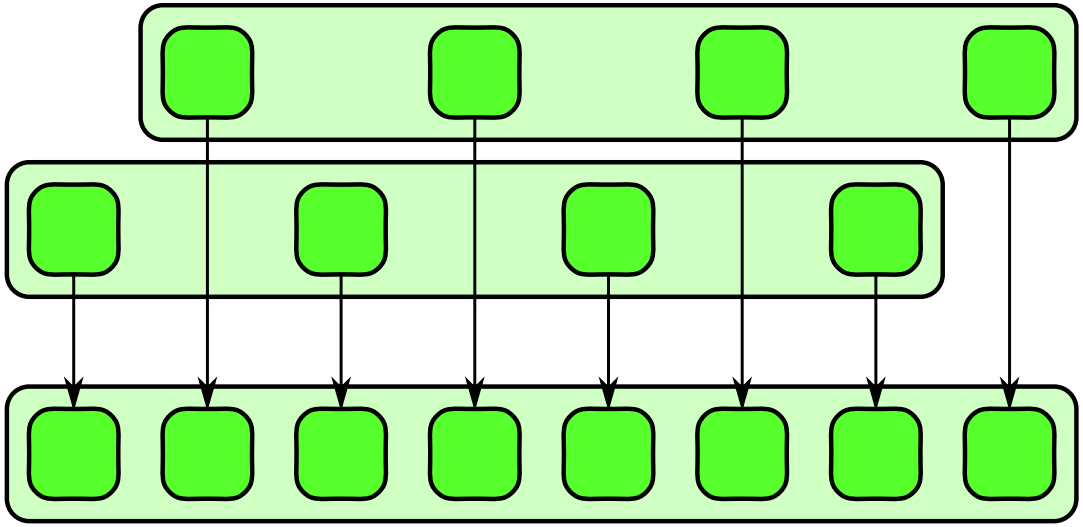
- Output data collection has the same number of elements as the number of indices in the index collection
- Elements of the output collection are the same type as the input data collection

Special Case of Gather: Shifts



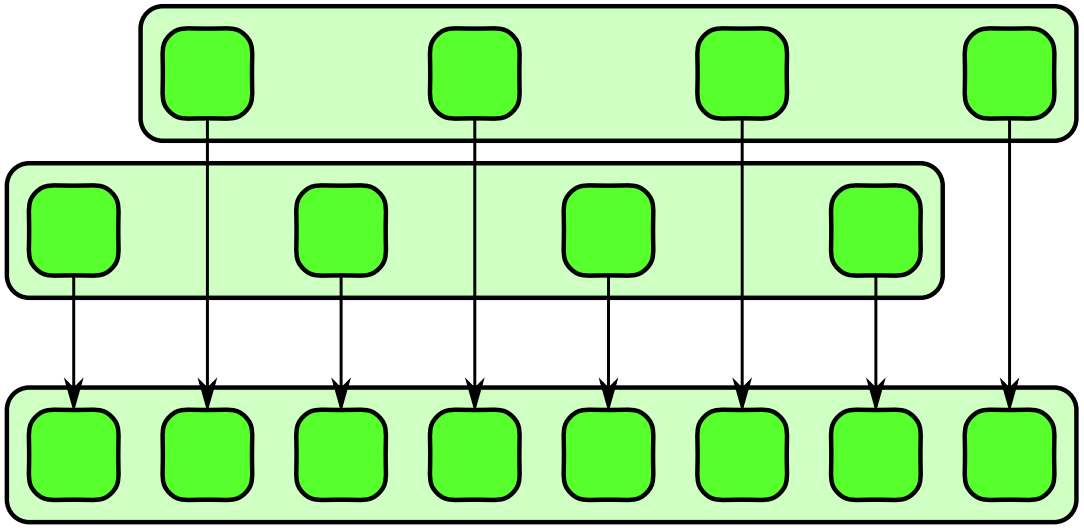
- Moves data to the left or right in memory
- Data accesses are offset by fixed distances

Special Case of Gather: Zip



- Interleaves data

Special Case of Gather: Zip



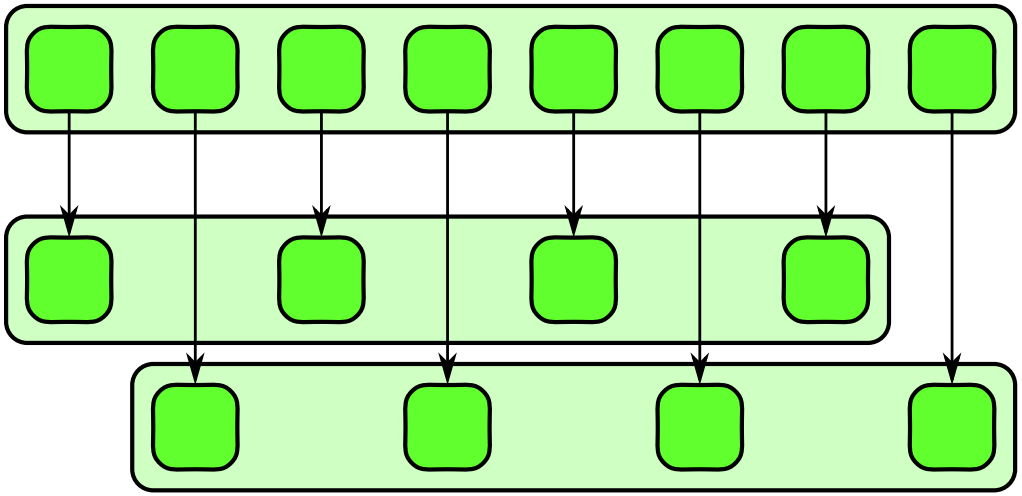
Array of Real Parts

Array of Imaginary Parts

Combined Sequence of
Real and Imaginary Parts

- Interleaves data
- Ex: Given two separate arrays of real parts and imaginary parts, use zip to combine them into a sequence of real and imaginary pairs.

Special Case of Gather: Unzip



- Reverses a zip, extracting subarrays at certain offsets and strides from an input array
- Ex: Given a sequence of complex numbers organized as pairs, extract real and imaginary parts into separate arrays

Gather vs. Scatter

Gather

- Combination of map with random **reads**
- Read locations provided as input

Scatter

- Combination of map with random **writes**
- Write locations provided as input
- Race conditions

Scatter: Serial Implementation

```
1  template<typename Data, typename Idx>
2  void scatter(
3      size_t n, // number of elements in output data collection
4      size_t m, // number of elements in input data and index collection
5      Data a[], // input data collection (m elements)
6      Data A[], // output data collection (n elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check output array bounds
12         A[j] = a[i]; // perform random write
13     }
14 }
```

Serial implementation of scatter in pseudocode

Scatter: Serial Implementation

```
1  template<typename Data, typename Idx>
2  void scatter(
3      size_t n, // number of elements in output data collection
4      size_t m, // number of elements in input data and index collection
5      Data a[], // input data collection (m elements)
6      Data A[], // output data collection (n elements)
7      Idx idx[] // input index collection (m elements)
8  ) {
9      for (size_t i = 0; i < m; ++i) {
10         size_t j = idx[i]; // get ith index
11         assert(0 <= j && j < n); // check output array bo
12         A[j] = a[i]; // perform random write
13     }
14 }
```

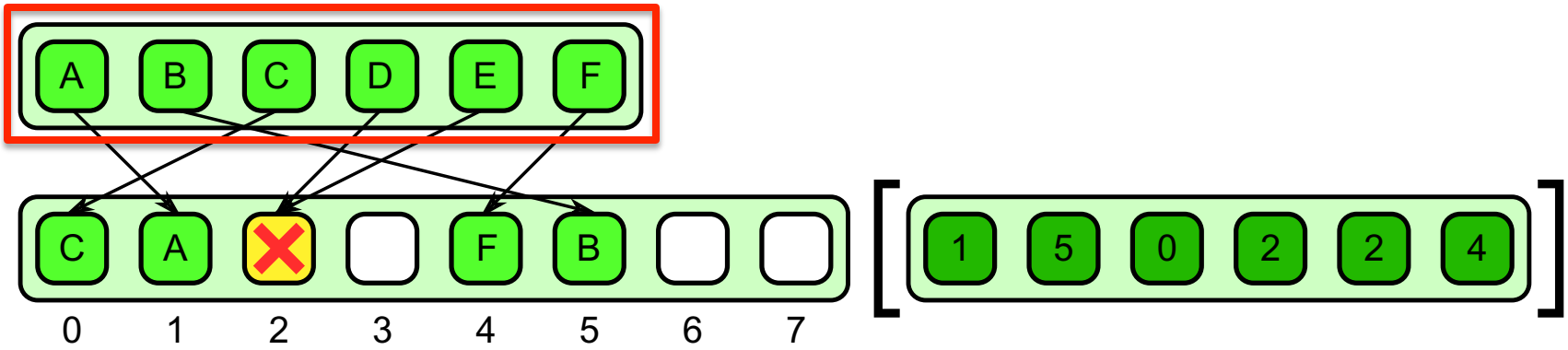
Parallelize over for loop to perform random write

Serial implementation of scatter in pseudocode

Scatter: Defined

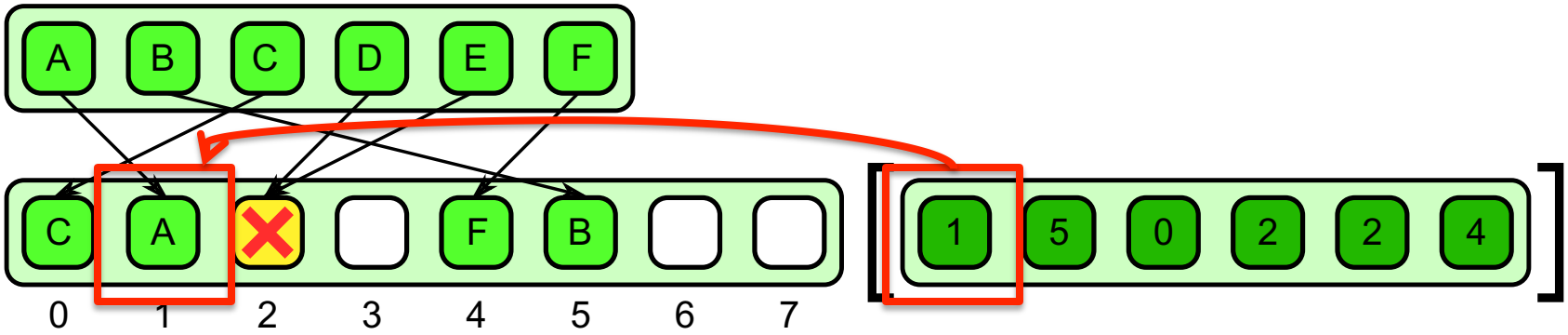
- Results from the combination of a map with a random write

Scatter: Defined



Collection of **input data**

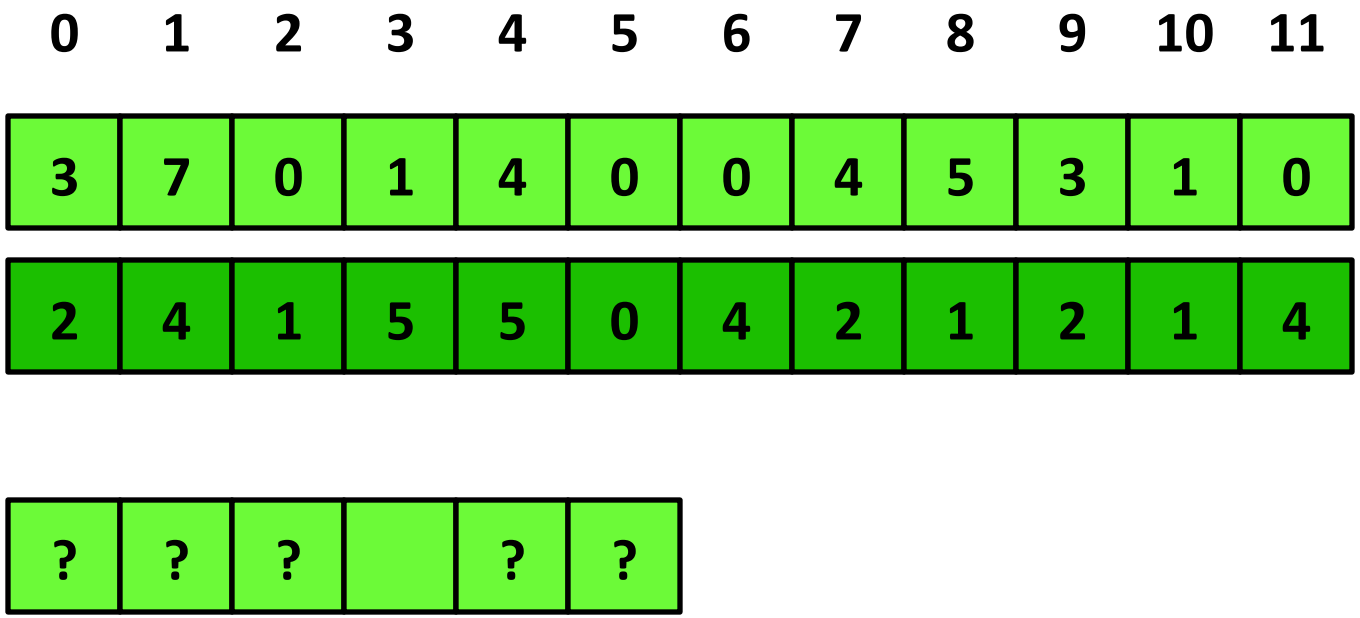
Scatter: Defined



Collection of input data written in parallel to the **write locations** specified.

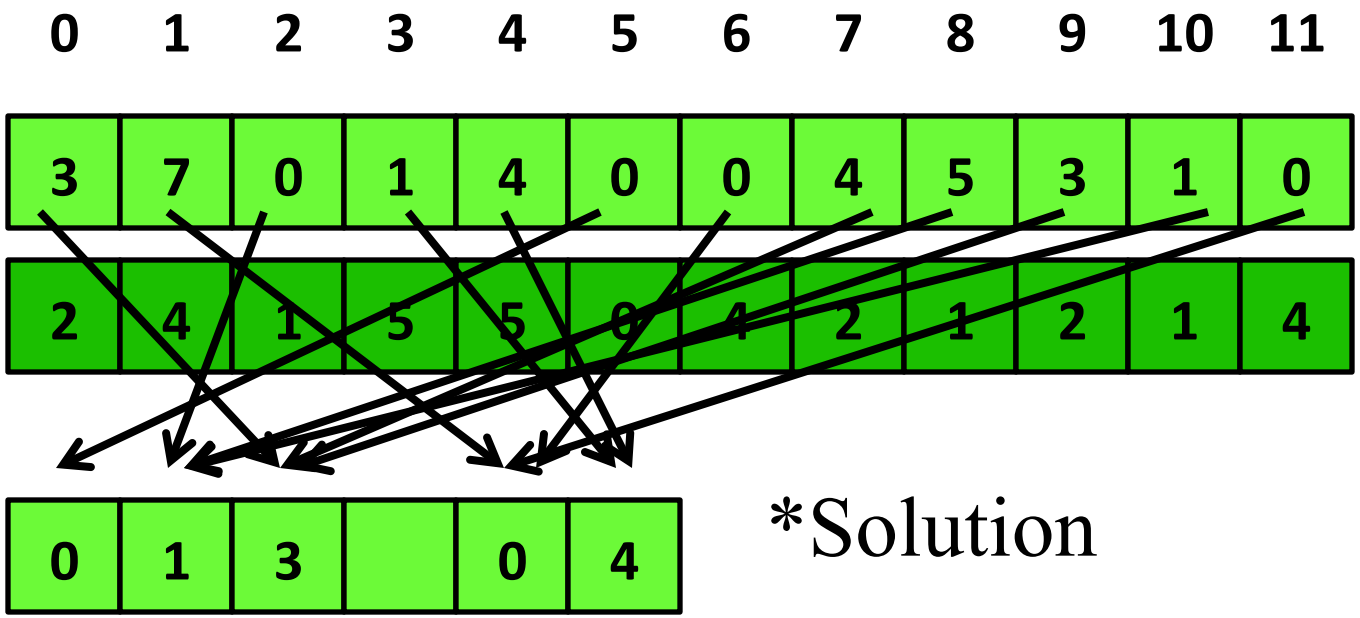
Quiz 2

Given the following locations and source array, what values should go into the input collection:



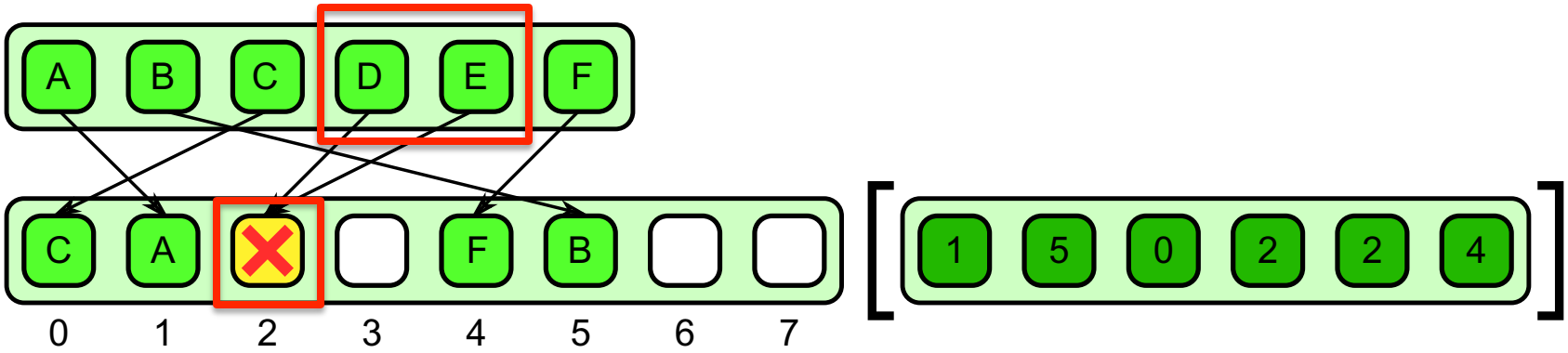
Quiz 2

Given the following locations and source array, what values should go into the input collection:



*Solution

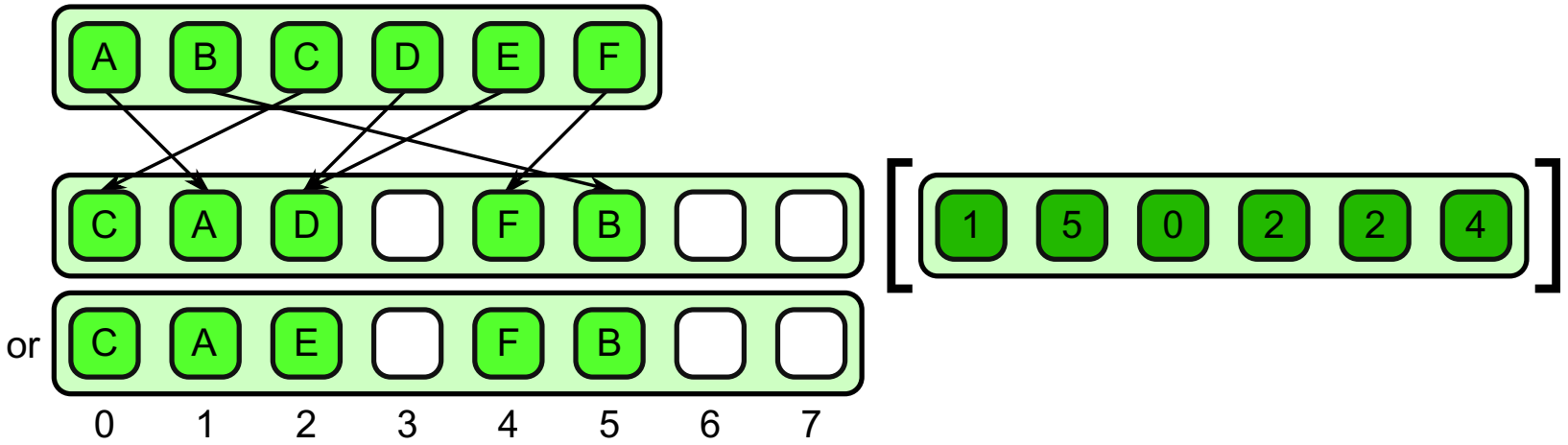
Scatter: Race Conditions



Collection of input data written in parallel to the **write locations** specified.

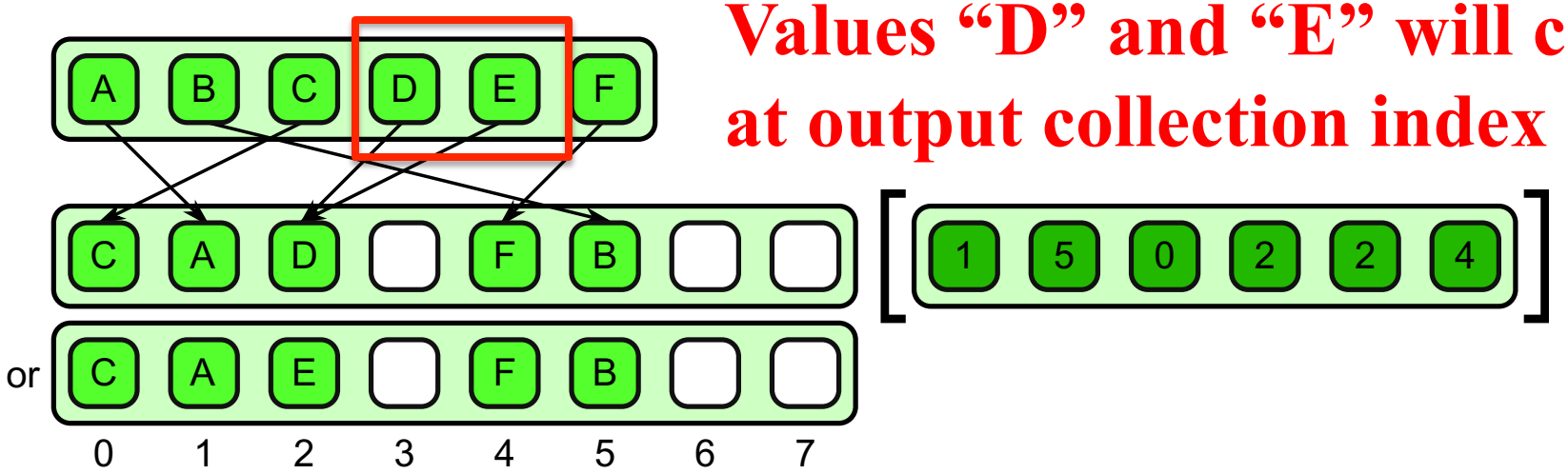
Race Condition: Two (or more) values being written to the same location in output collection. Unclear what the result should be. **Need rules to resolve collisions!**

Collision Resolution: Atomic Scatter



- **Non-deterministic** approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

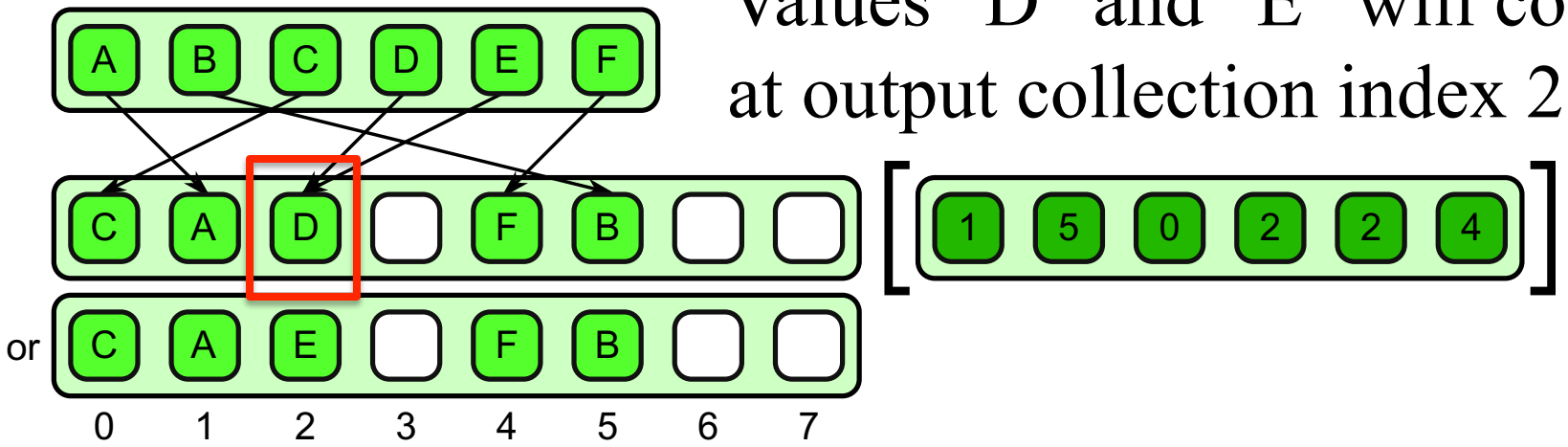
Collision Resolution: Atomic Scatter



Values "D" and "E" will collide at output collection index 2.

- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

Collision Resolution: Atomic Scatter

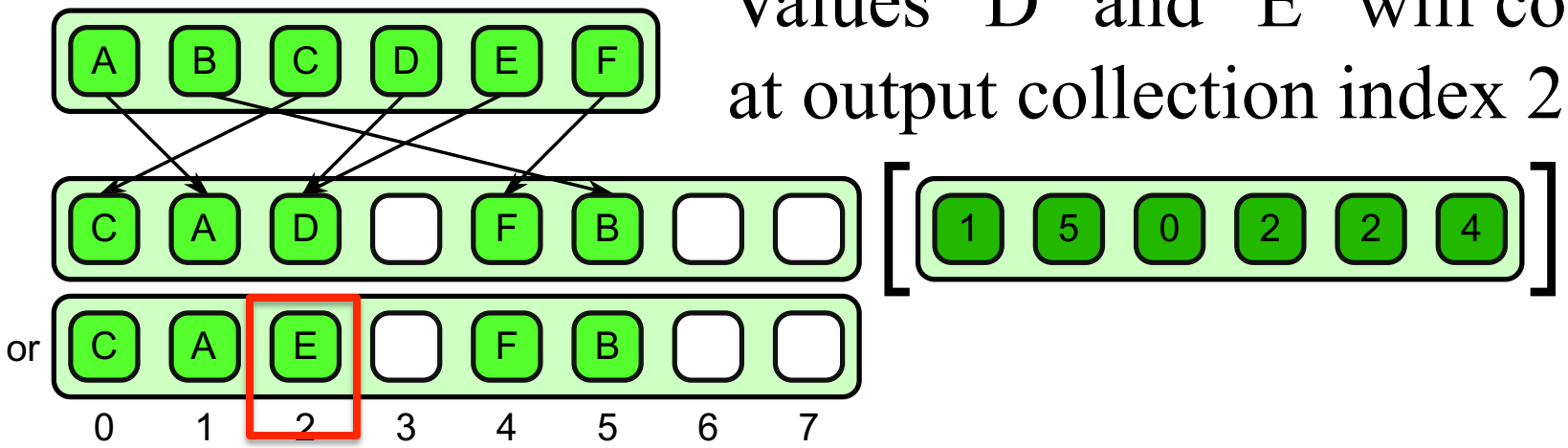


Values "D" and "E" will collide at output collection index 2.

- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Either "D"…

Collision Resolution: Atomic Scatter

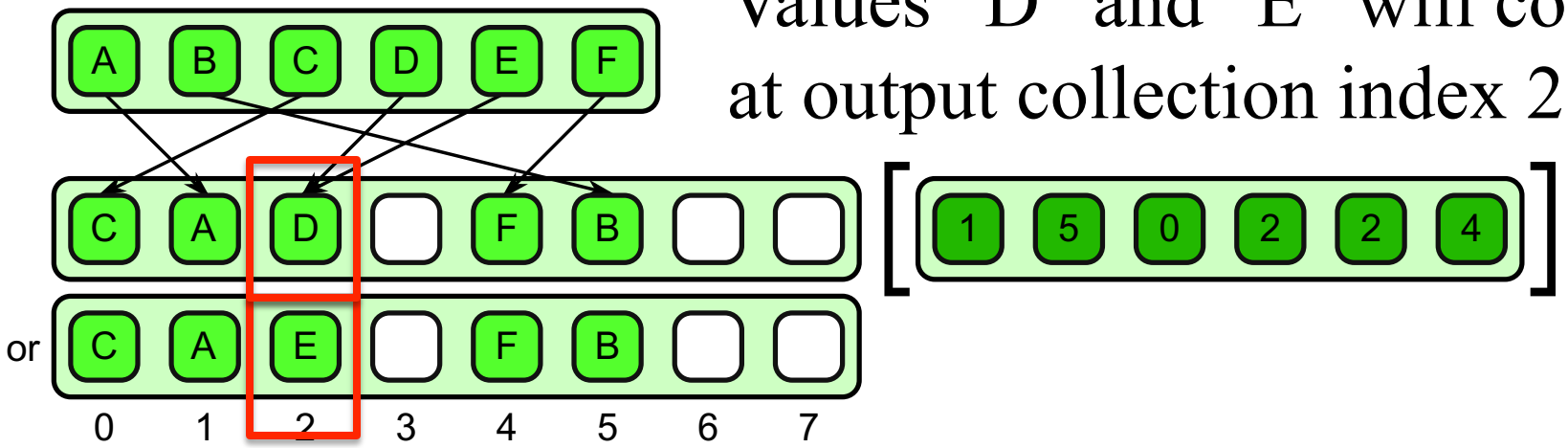


Values "D" and "E" will collide at output collection index 2.

- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Either "D" or "E"…

Collision Resolution: Atomic Scatter

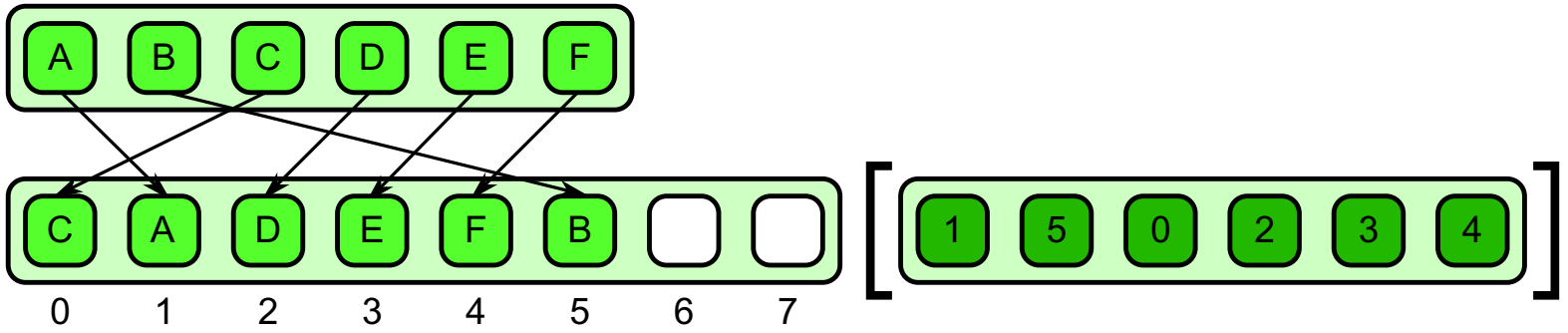


Values "D" and "E" will collide at output collection index 2.

- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

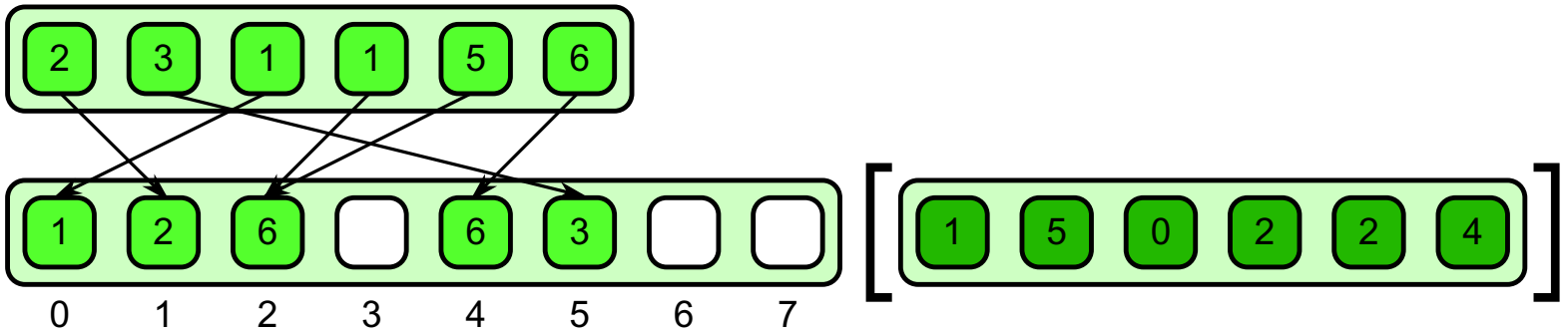
Either "D" or "E" will be written to index 2 of output collection

Collision Resolution: Permutation Scatter



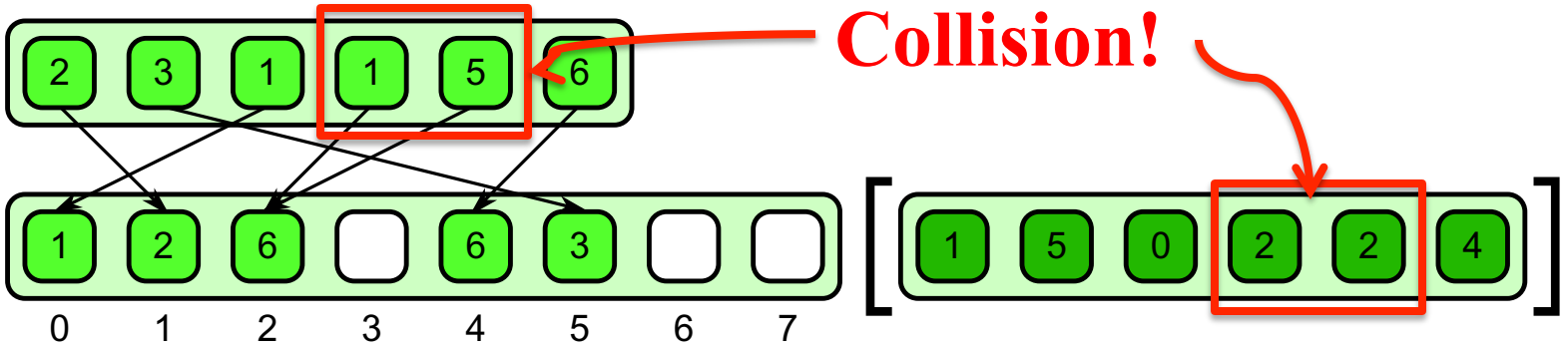
- Collisions are **illegal**
- Check for collisions in advance → turn scatter into gather
- Ex: FFT scrambling, matrix/image transpose, unpacking

Collision Resolution: Merge Scatter



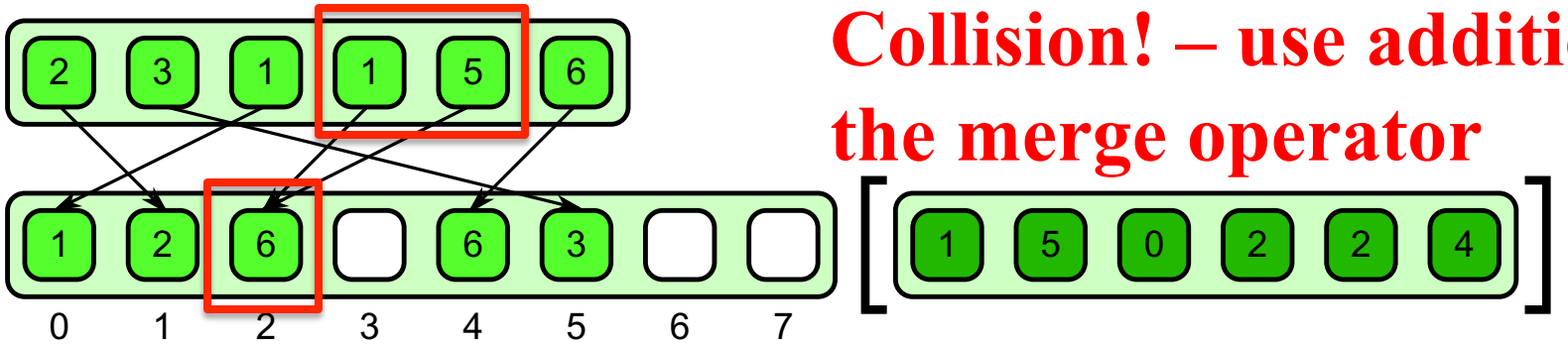
- Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Merge Scatter



- Associative and commutative operators are provided to merge elements in case of a collision

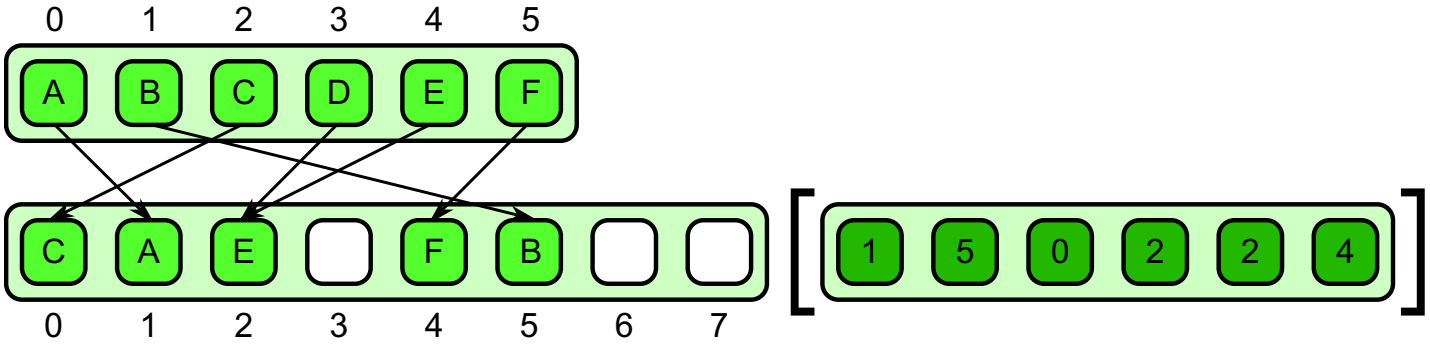
Collision Resolution: Merge Scatter



Collision! – use addition as the merge operator

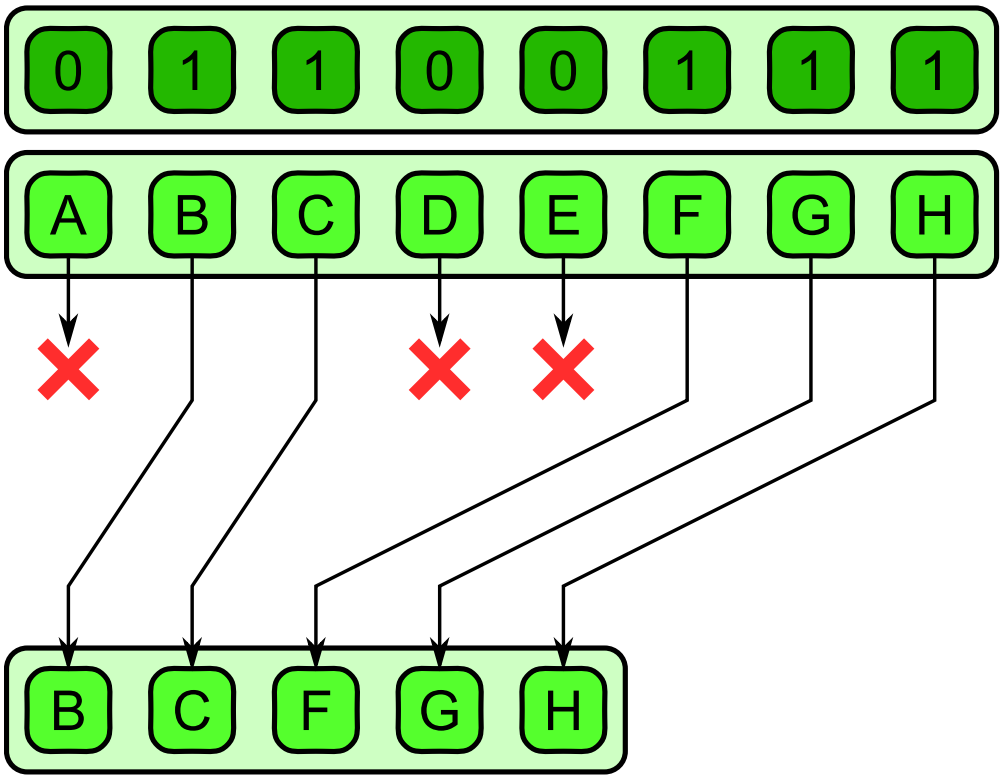
- Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Priority Scatter



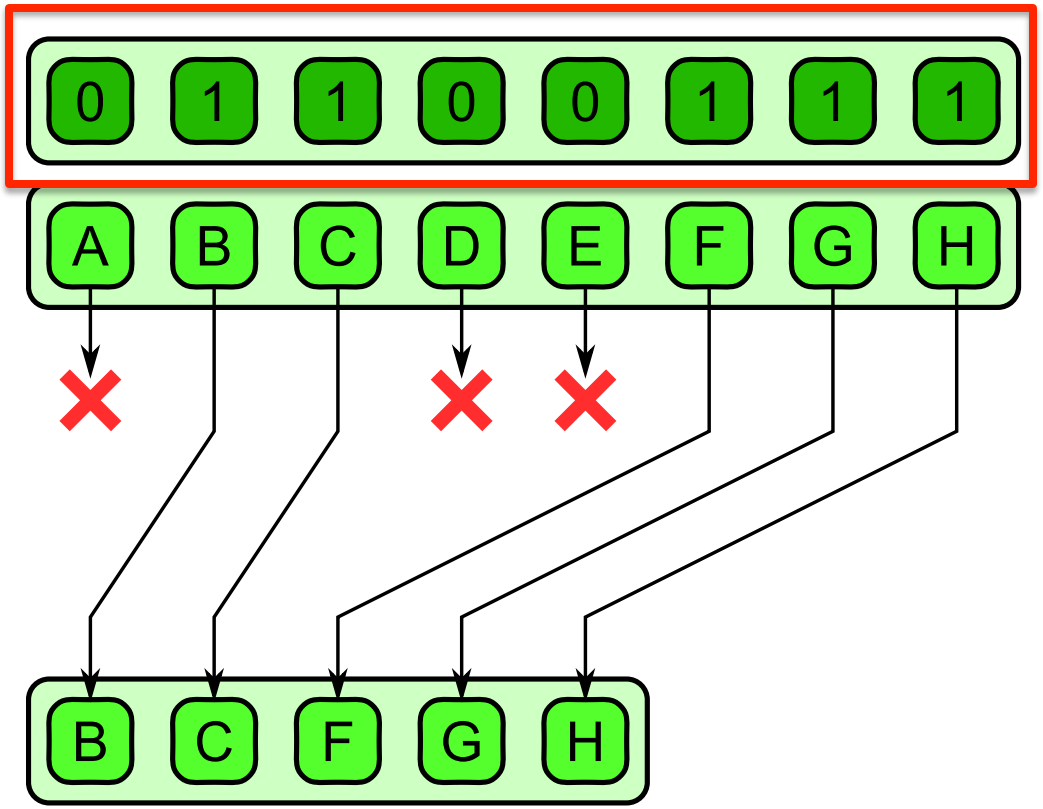
- Every element in the input array is assigned a priority based on its position
- Priority is used to decide which element is written in case of a collision
- Ex: 3D graphics rendering

Pack: Defined



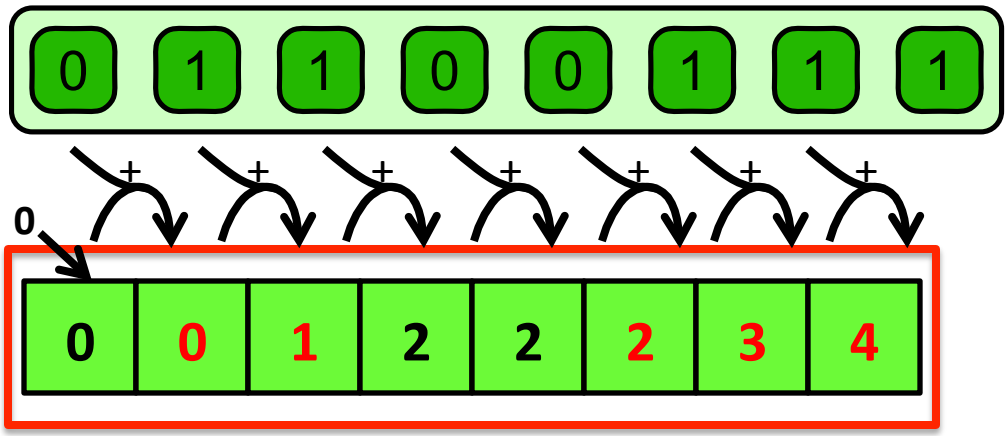
- Used to eliminate unused elements from a collection
- Retained elements are moved so they are contiguous in memory → performance improvement

Pack Algorithm



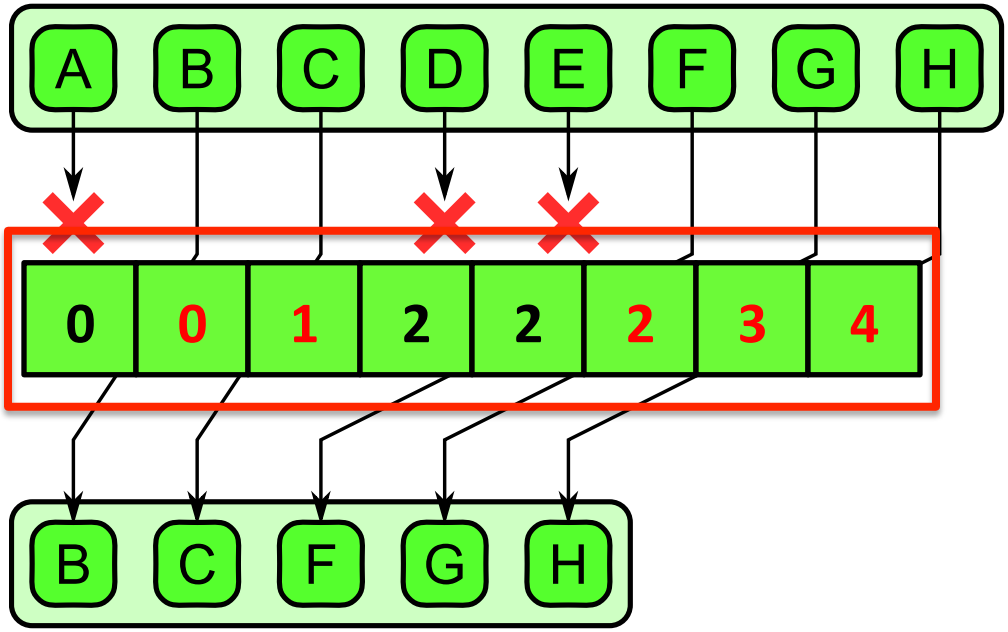
1. **Convert input array of Booleans into integer 0's and 1's**

Pack Algorithm



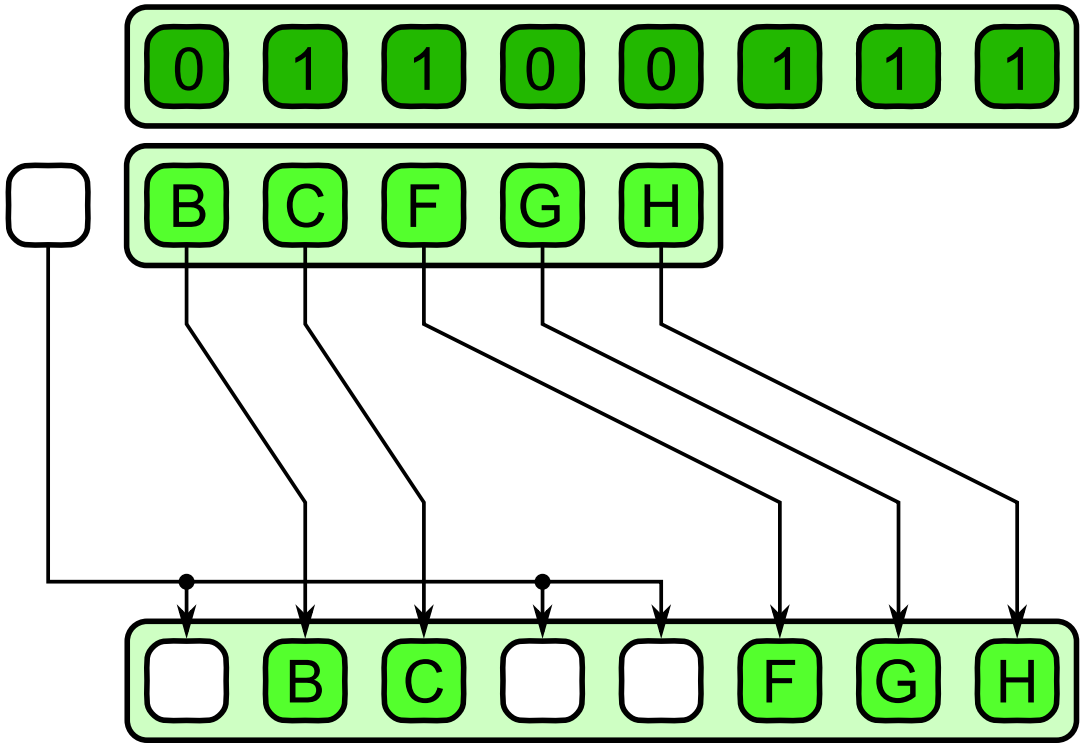
1. Convert input array of Booleans into integer 0's and 1's
2. **Exclusive scan of this array with the addition operation**

Pack Algorithm



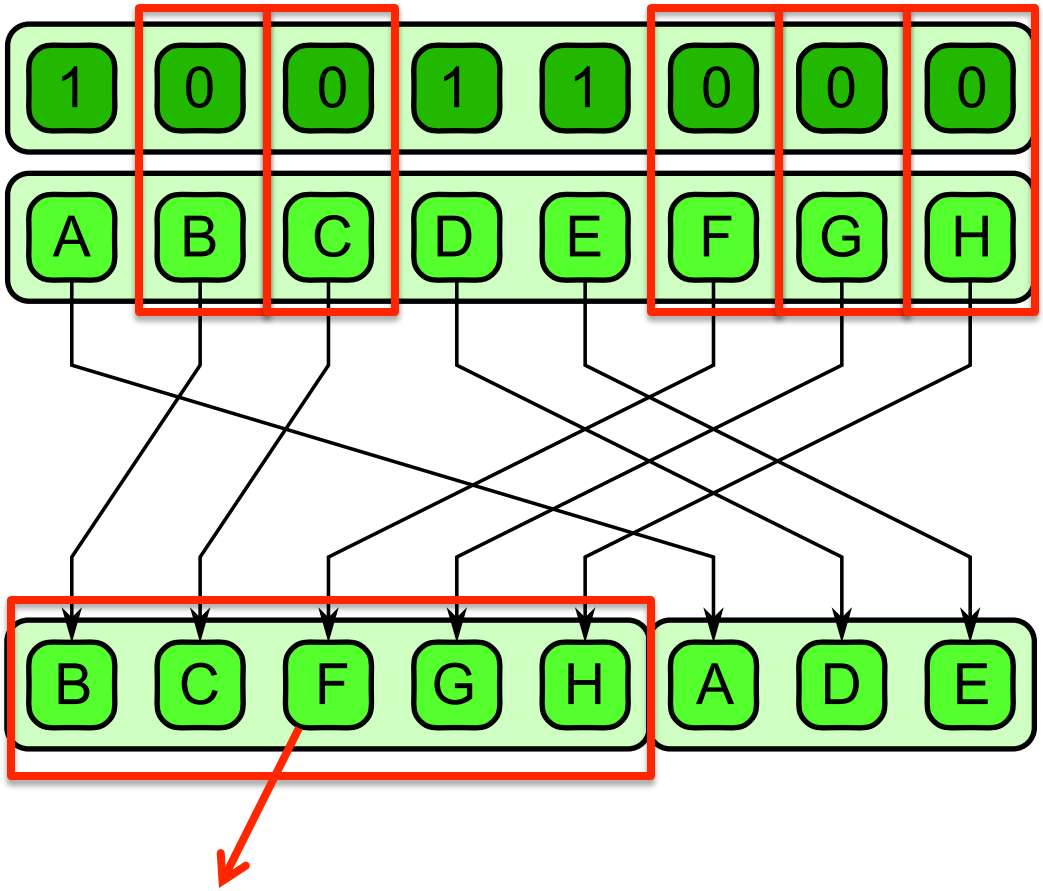
1. Convert input array of Booleans into integer 0's and 1's
2. Exclusive scan of this array with the addition operation
3. **Write values to output array based on offsets**

Unpack Defined



- Inverse of pack operation
- Given the same data on which elements were kept and which were discarded, can place elements back in their original locations

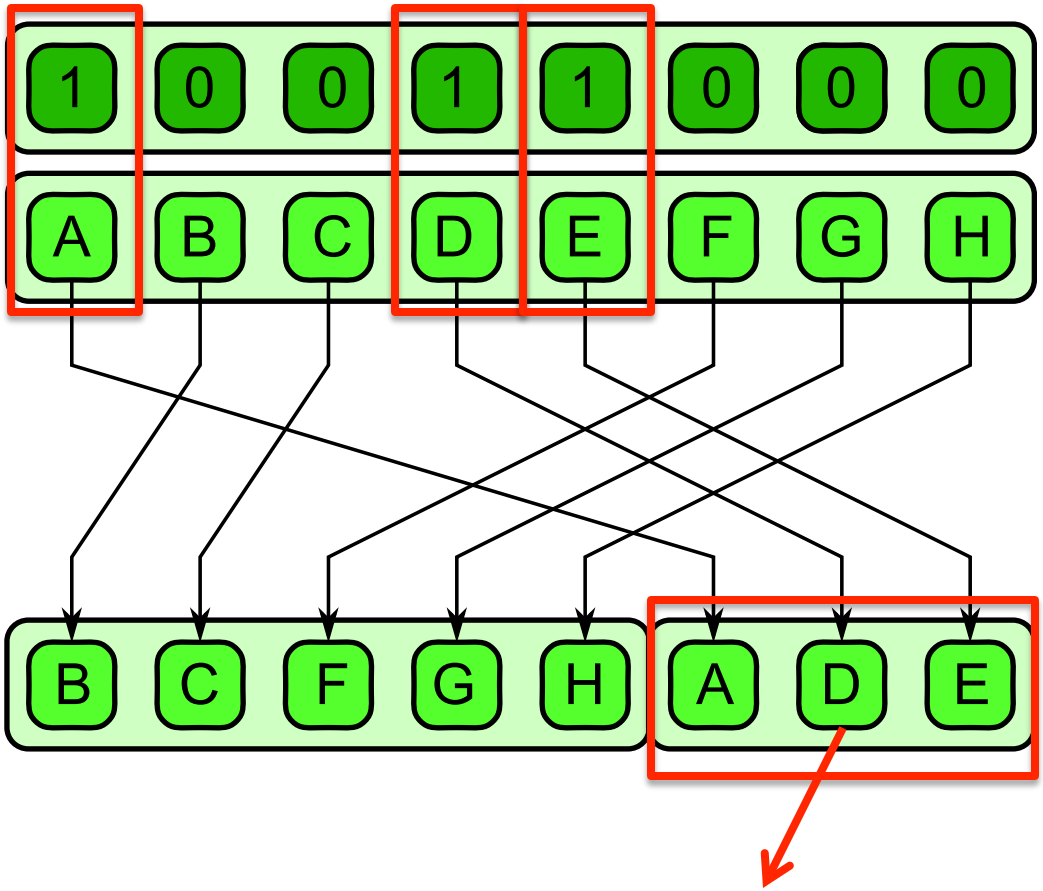
Generalization of Pack: Split



Upper half of output collection: values equal to 0

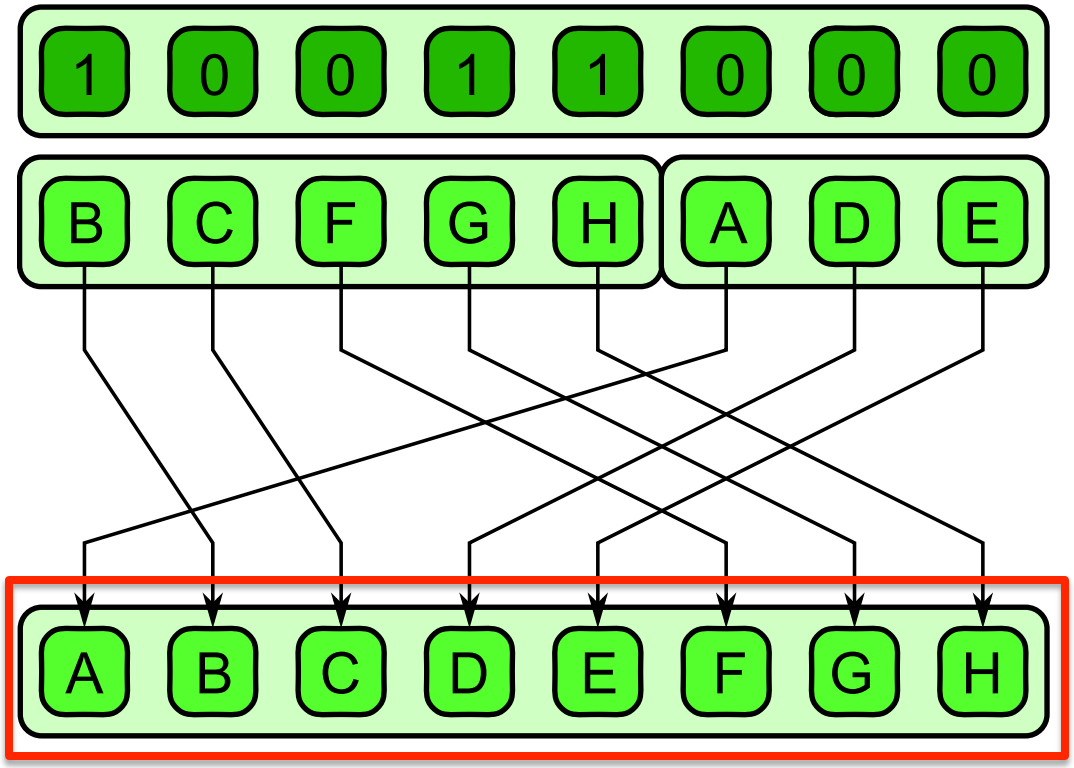
- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

Generalization of Pack: Split



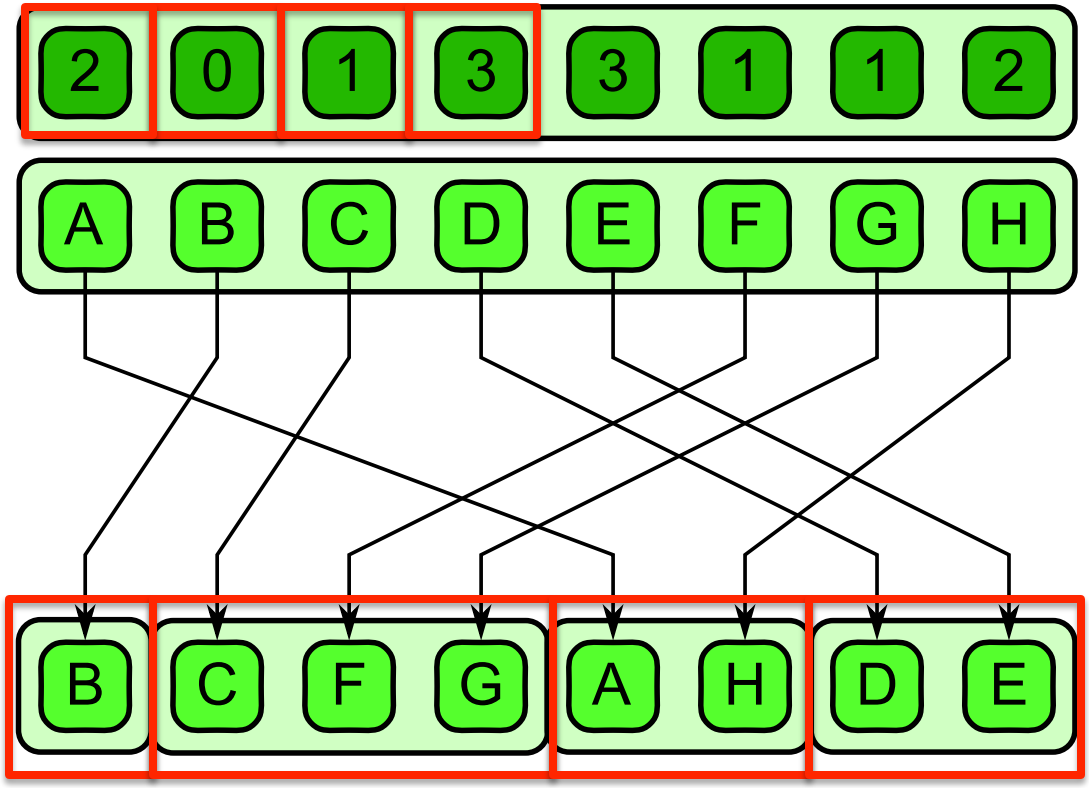
- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

Generalization of Pack: Unsplit



- Inverse of split
- Creates **output collection** based on original input collection

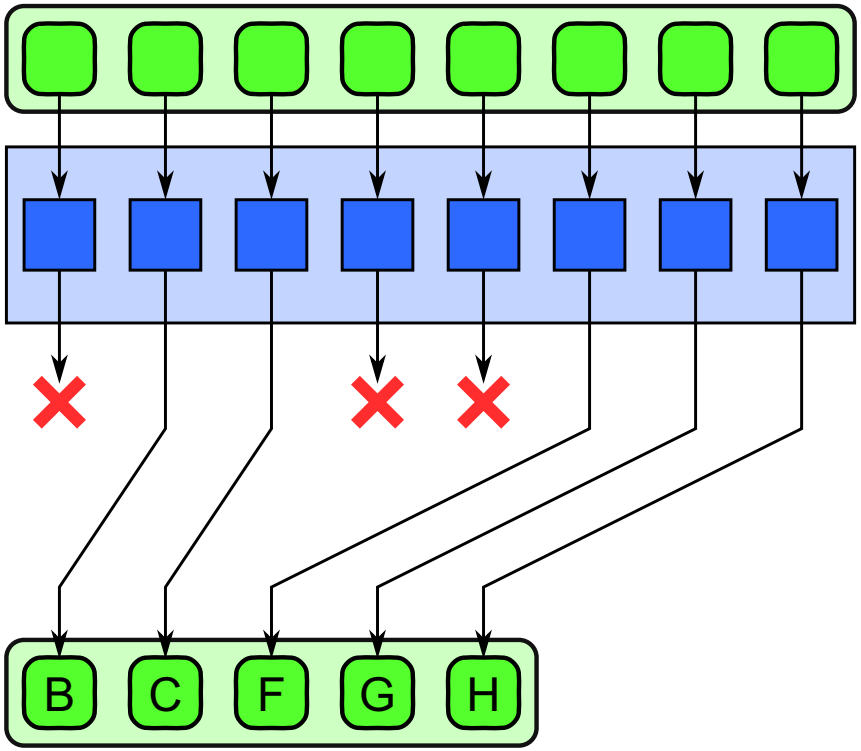
Generalization of Pack: Bin



- Generalized split to support more than two categories
- Ex: radix sort, pattern classification

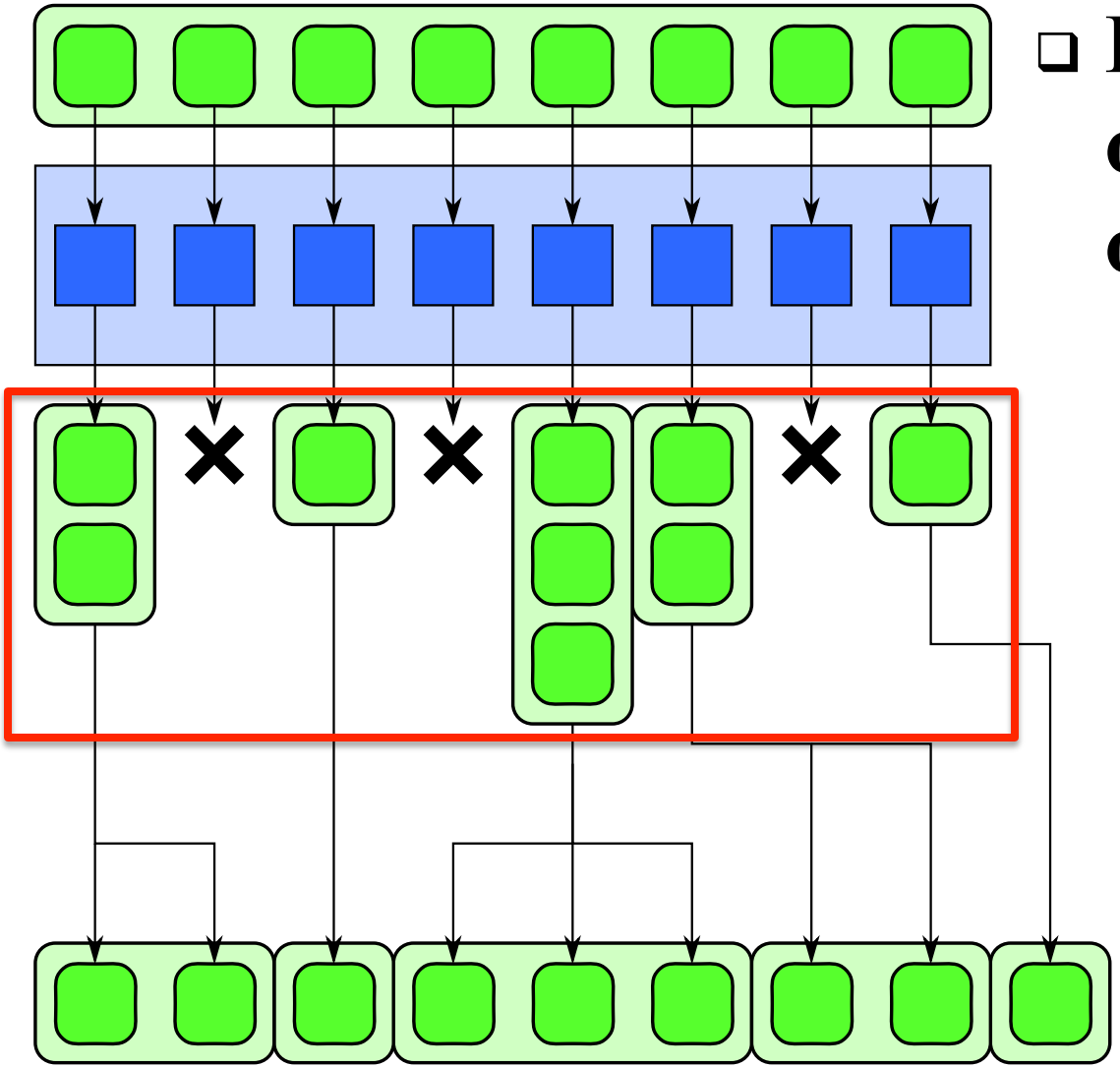
4 different categories = 4 bins

Fusion of Map and Pack



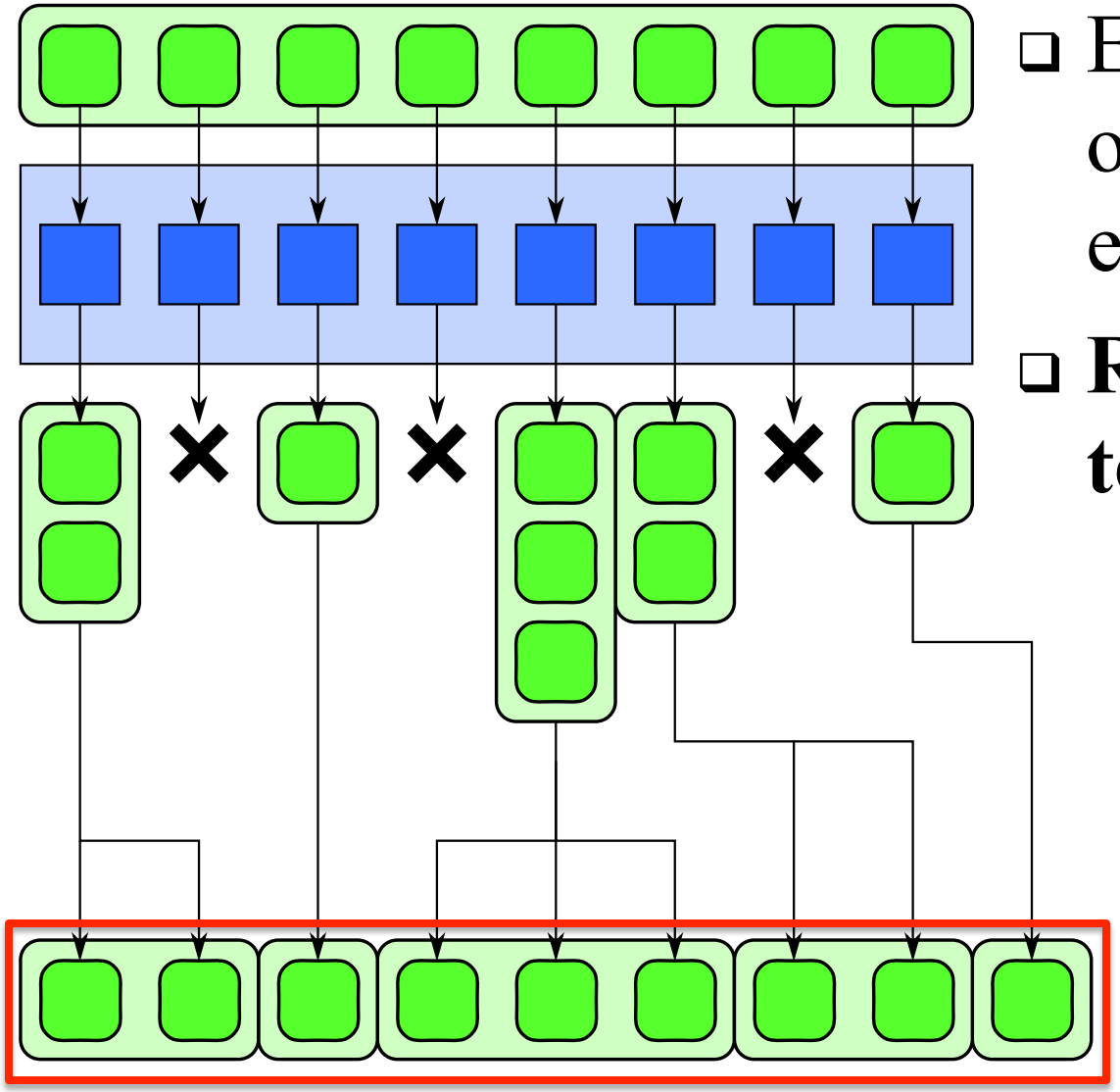
- Advantageous if most of the elements of a map are discarded
- **Map** checks pairs for collision
- **Pack** stores only actual collisions
- Output BW ~ results reported, not number of pairs tested
- Each element can output 0 or 1 element

Generalization of Pack: Expand



- **Each element can output any number of elements**

Generalization of Pack: Expand



- Each element can output any number of elements
- **Results are fused together in order**

Parallelizing Algorithms

- Common strategy:
  1. Divide up the computational domain into sections
  2. Work on the sections individually
  3. Combine the results

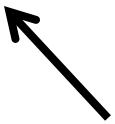
Discussed in ch 8

Methods: divide-and-conquer, fork-join, geometric decomposition, partitions, segments

Parallelizing Algorithms

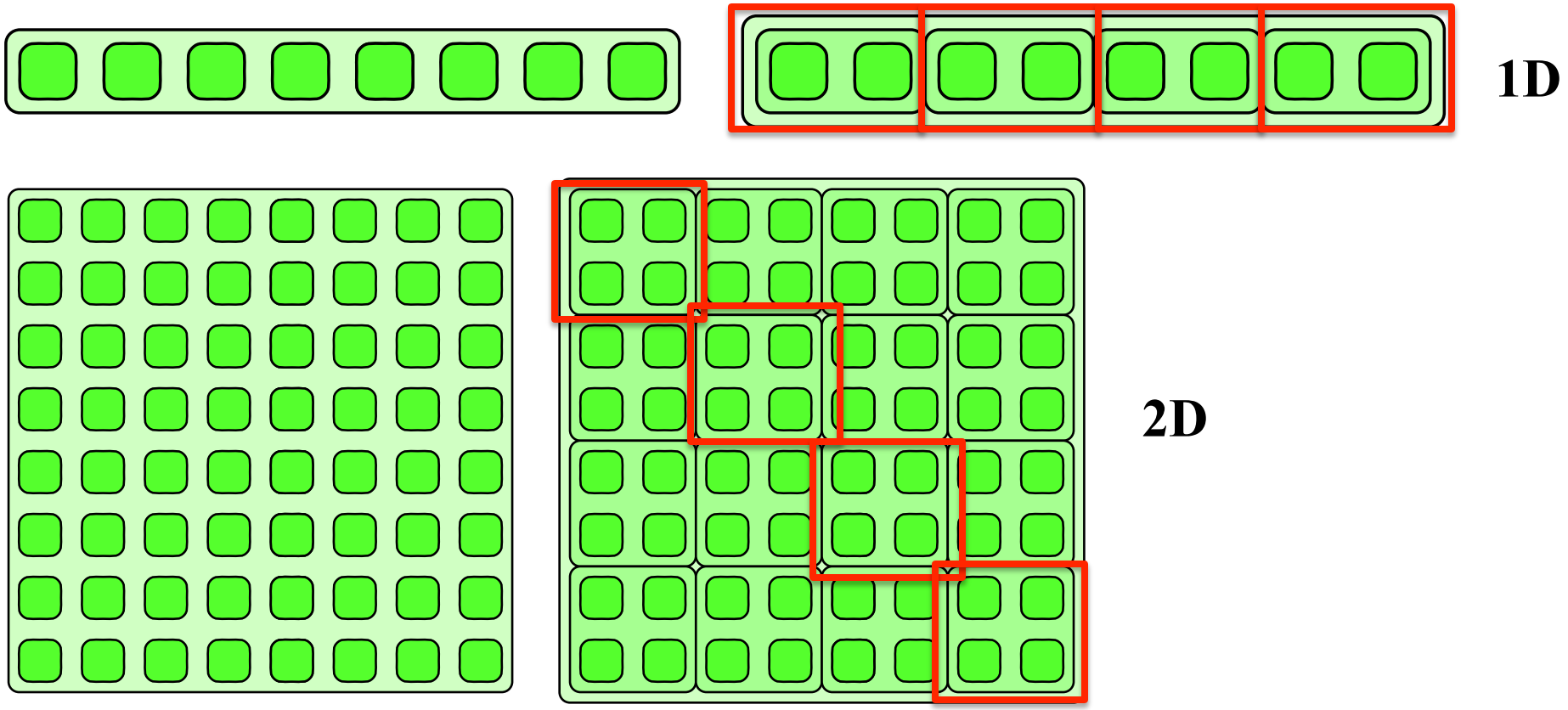
- Common strategy:
  1. Divide up the computational domain into sections
  2. Work on the sections individually
  3. Combine the results

Methods: divide-and-conquer, fork-join, geometric decomposition, **partitions**, segments



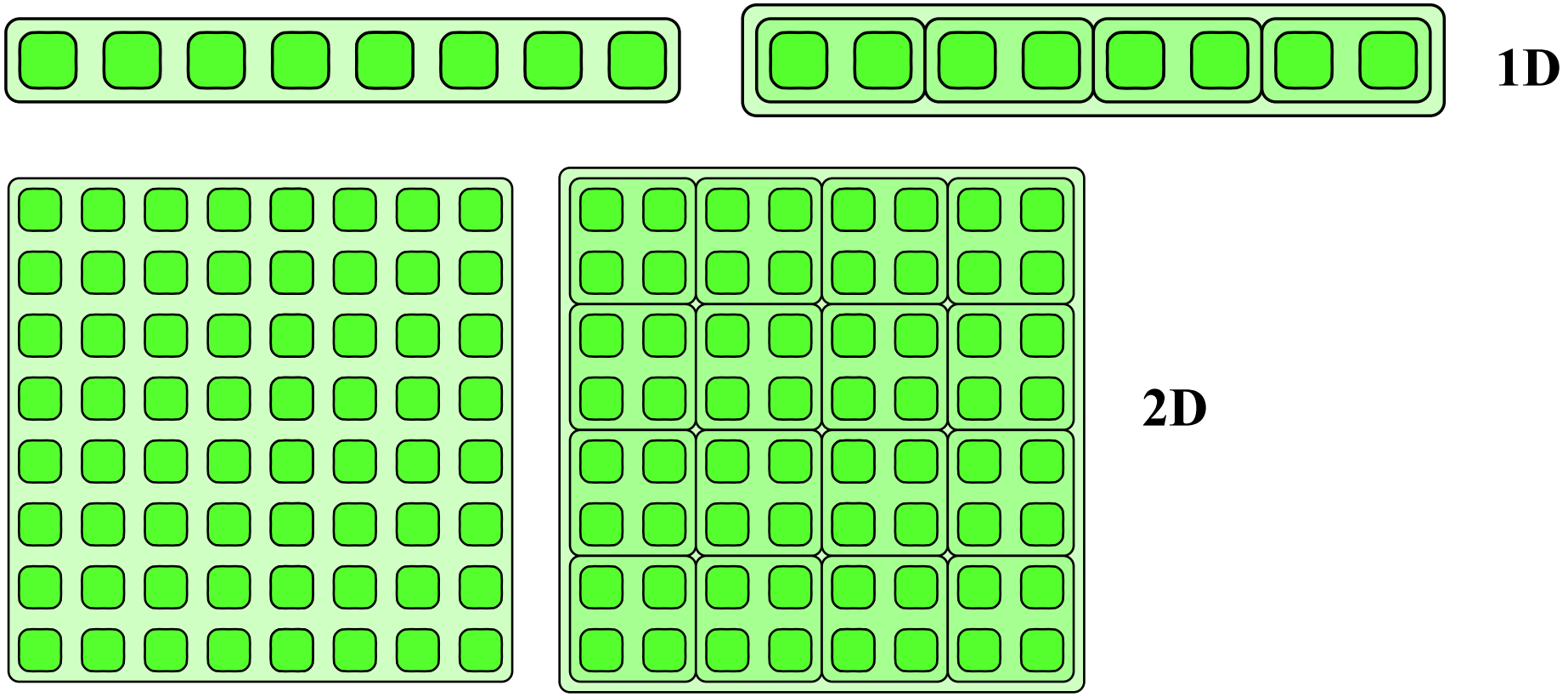
Discussed next…

Partitioning



- Data is divided into **non-overlapping**, **equal-sized** regions

Partitioning



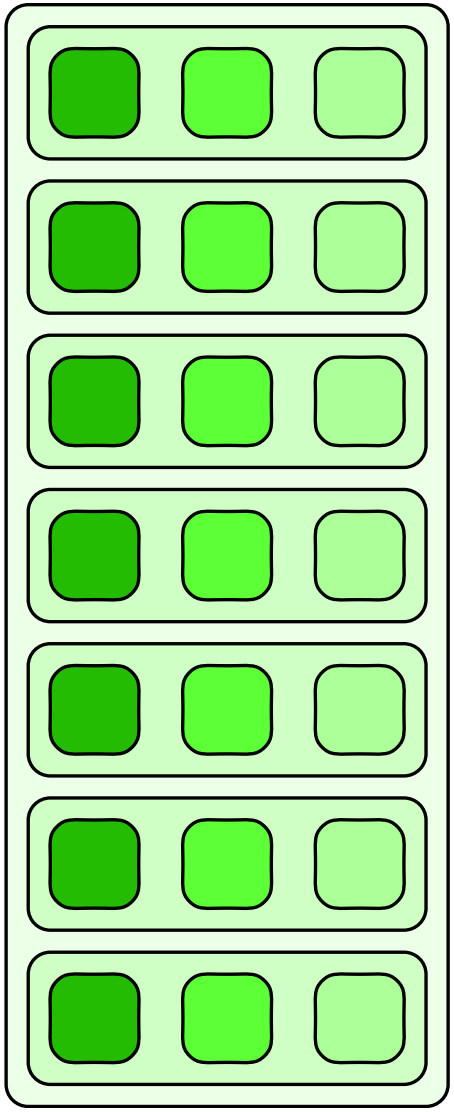
- Data is divided into **non-overlapping**, equal-sized regions

Avoid write conflicts and race conditions

Segmentation

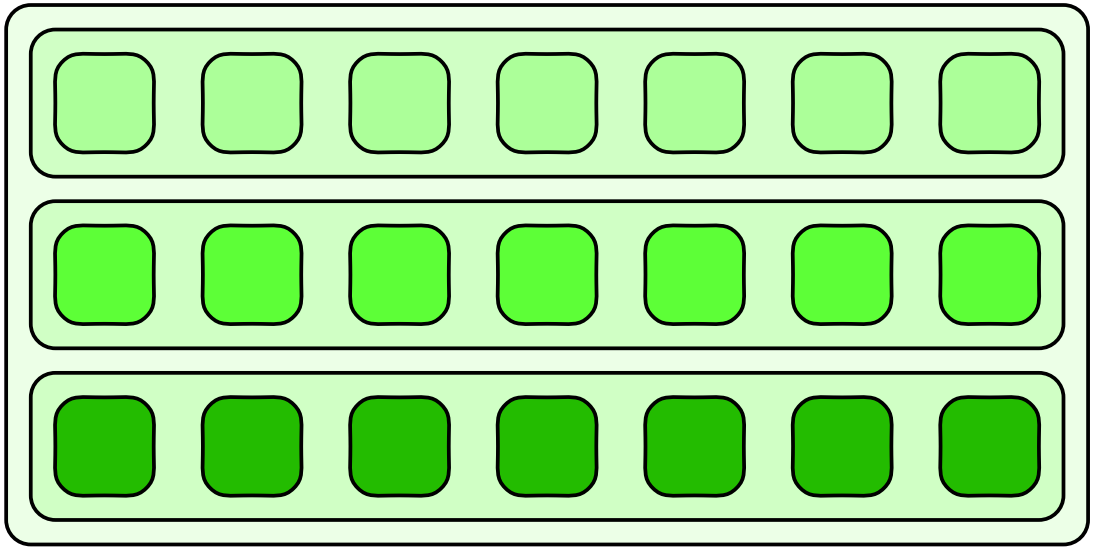
- Data is divided into **non-uniform** non-overlapping regions
- Start of each segment can be marked using:
  - Array of integers
  - Array of Boolean flags

Array of Structures (AoS)



- May lead to better cache utilization if data is accessed randomly

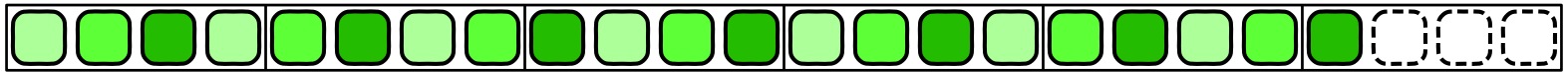
Structures of Arrays (SoA)



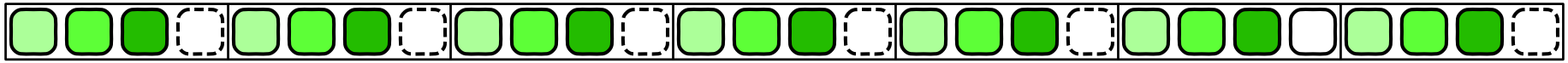
- Typically better for vectorization and avoidance of false sharing

Data Layout Options

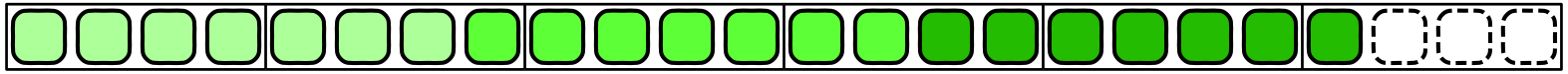
Array of Structures (AoS), padding at end



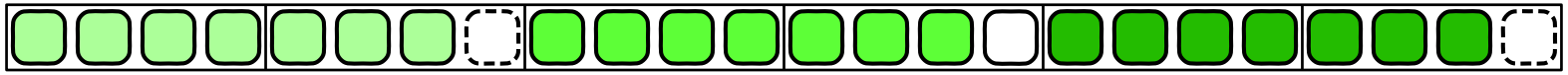
Array of Structures (AoS), padding after each structure



Structure of Arrays (SoA), padding at end



Structure of Arrays (SoA), padding after each component



Example Implementation

AoS Code

```
struct node {
  float x, y, z;
};
struct node NODES[1024];

float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

SoA Code

```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

AoS Code

```
struct node {
  float x, y, z;
};
struct node NODES[1024];

float dist[1024];
for(i=0;i<1024;i+=16){
   float x[16],y[16],z[16],d[16];
   x[:] = NODES[i:16].x;
   y[:] = NODES[i:16].y;
   z[:] = NODES[i:16].z;
   d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
   dist[i:16] = d[:];
}
```

- Most logical data organization layout
- Extremely difficult to access memory for reads (gathers) and writes (scatters)
- Prevents efficient vectorization

SoA Code

```
struct node1 {
  float x[1024], y[1024], z[1024];
}
struct node1 NODES1;

float dist[1024];
for(i=0;i<1024;i+=16){
  float x[16],y[16],z[16],d[16];
  x[:] = NODES1.x[i:16];
  y[:] = NODES1.y[i:16];
  z[:] = NODES1.z[i:16];
  d[:] = sqrtf(x[:]*x[:] + y[:]*y[:] + z[:]*z[:]);
  dist[i:16] = d[:];
}
```

- Separate arrays for each structure-field keeps memory accesses contiguous when vectorization is performed over structure instances