CS 261: Data Structures

Sorted Dynamic Array Bag and Set

Ordered Collections

- How do we organize data in dictionaries or phonebooks?

- Find in a phonebook:
  - the phone number of John Smith
  - the person with phone number 753-6692

Guess My Number

- Integer numbers are ordered
- I'm thinking of a number in [1, 100]
- Ask questions to guess my number

Binary Search

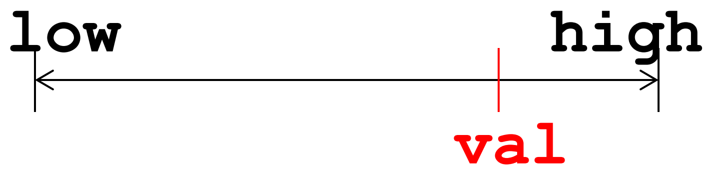
- The formal name -- Binary Search
- Works by iteratively dividing the interval which contains the value
- Dividing, e.g., in half, in each step
- Suppose we have n items, how many iterations before the interval is of size one?

Log n search

- A **`log n`** search is **much much** faster than an **`O(n)`** search.

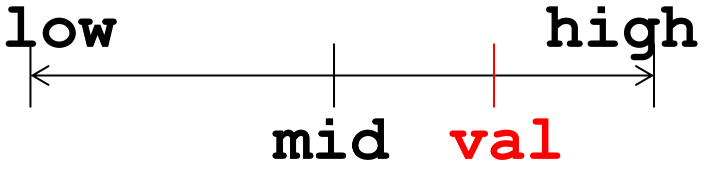
Binary Search Algorithm

```
int binarySearch (TYPE * data, int size, TYPE val){
   int low = 0;/*index*/
   int high = size; /*index*/

   while (low < high) {

         ...

   }
   return  ???; /*returns the index of val in data*/
}
```



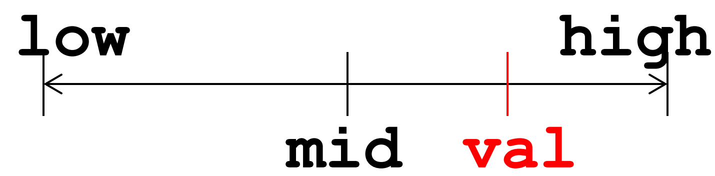
Binary Search Algorithm

```
int binarySearch (TYPE * data, int size, TYPE val){
   int low = 0;
   int high = size;

   while (low < high) {
     int mid = (low + high) / 2;


    ...
   }
   return    ??? ;
}
```



Binary Search Algorithm

```
int binarySearch (TYPE * data, int size, TYPE val){
   ...
   while (low < high) {
     int mid = (low + high) / 2;
     if (data[mid] < val)
         low = mid + 1;
      else
         high = mid;
   }
   return    ???? ;
}
```



Should we return **low** or **high**?

Binary Search Algorithm

```
int binarySearch (TYPE * data, int size, TYPE val){
   ...
   while (low < high) {
     int mid = (low + high) / 2;
     if (data[mid] < val))
         low = mid + 1;
      else
         high = mid;
   }
   return low;
}
```

What does this Algorithm Return

- If value is found, returns its **index**
- If value is not found, returns **index** *where it can be inserted* without violating ordering
- **Careful**: returned index can be **larger than the size** of a collection

Makes which Bag operation faster?

- Suppose we use the dynamic array implementation of a bag
- Which operation is made faster by using a binary search?
  - Add(element)
  - Contains(element)
  - Remove(element)

An example operation

```
int sortedContains (struct dynArr *da, TYPE val)
{
    int idx = binarySearch(da->data, da->size, val);
    if (idx < da->size && da->data[idx] == val)
        return 1;
    return 0;
}
```

O(log n)

Add to a sorted Dynamic Array

```
int sortedAdd (struct dynArr *da, TYPE val)
{
   /* find where to insert the new element */
   int idx = binarySearch(da->data, da->size, val);
   ...
}
```

Add to a sorted Dynamic Array

```
int sortedAdd (struct dynArr *da, TYPE val)
{
   int idx = binarySearch(da->data, da->size, val);
   _addAt(da, idx, val);
}
```

Complexity?

Add to a sorted Dynamic Array

```
int sortedAdd (struct dynArr *da, TYPE val)
{
   int idx = binarySearch(da->data, da->size, val);
   _addAt(da, idx, val);
}
```

$$O(\log n) + O(n) = O(n)$$

Remove

```
int sortedRemove (struct dynArr *da, TYPE val) {
   int idx = binarySearch(da->data, da->size, val);
   if (idx < da->size && da->data[idx] == val)
       _removeAt(da, idx);
}
```

Complexity?

Remove

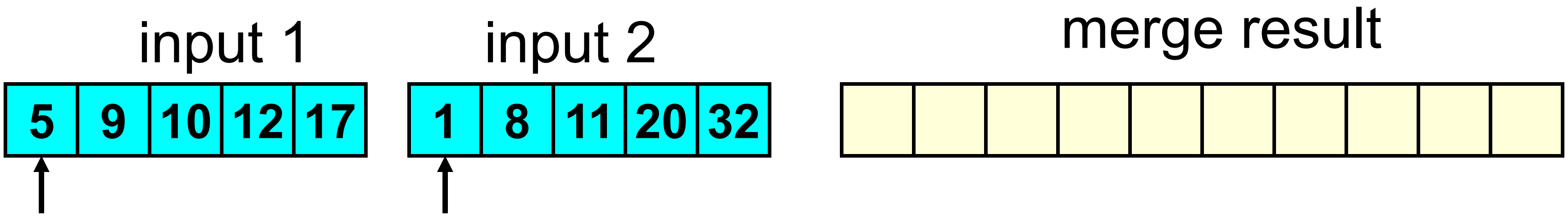
```
int sortedRemove (struct dynArr *da, TYPE val) {
   int idx = binarySearch(da->data, da->size, val);
   if (idx < da->size && da->data[idx] == val)
      _removeAt(da, idx);
}
```

$$O(\log n) + O(n) = O(n)$$

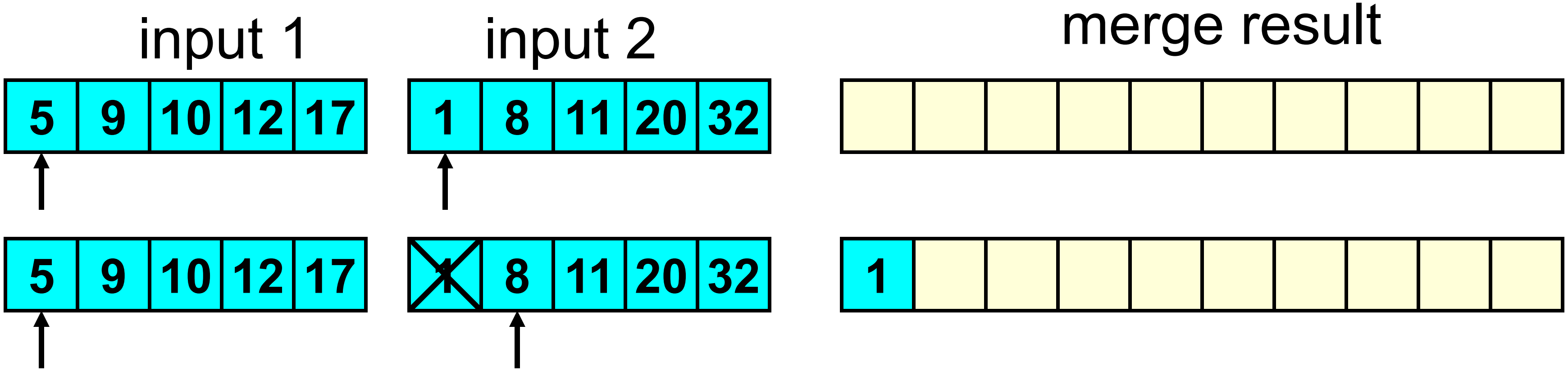
Why else do we deed an ordered collection?

- Fast merge operations
- Fast set operations (special case)
  - union
  - intersection

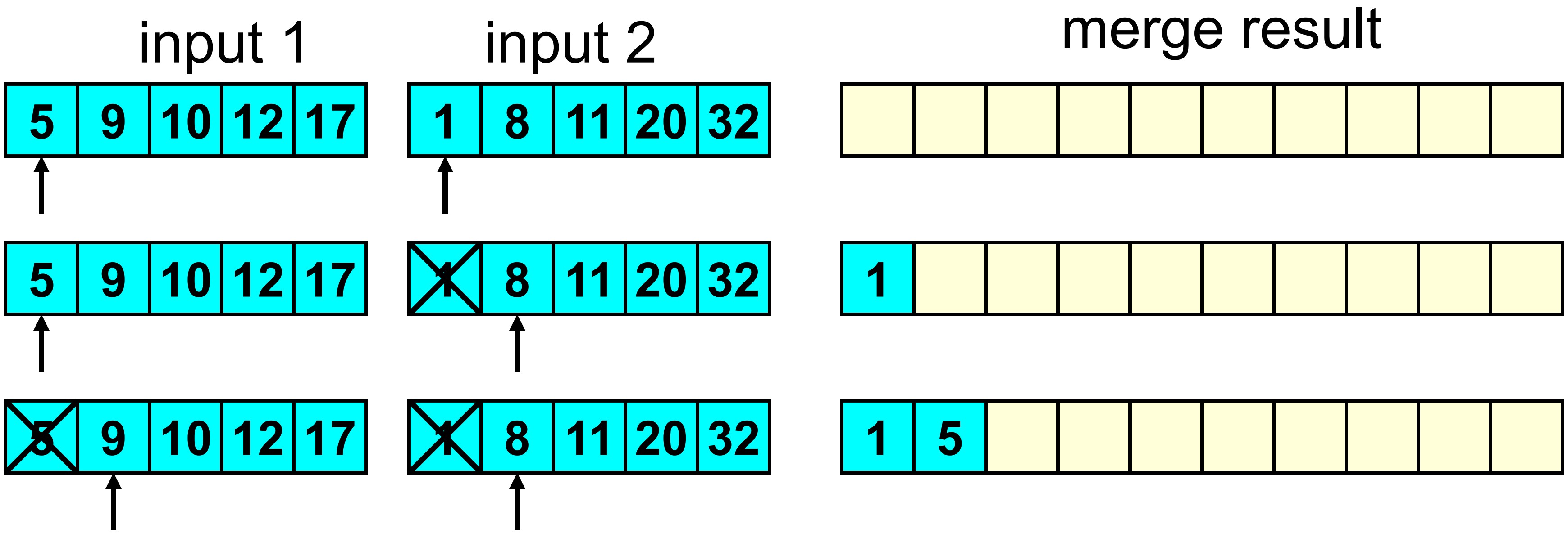
Fast Merge



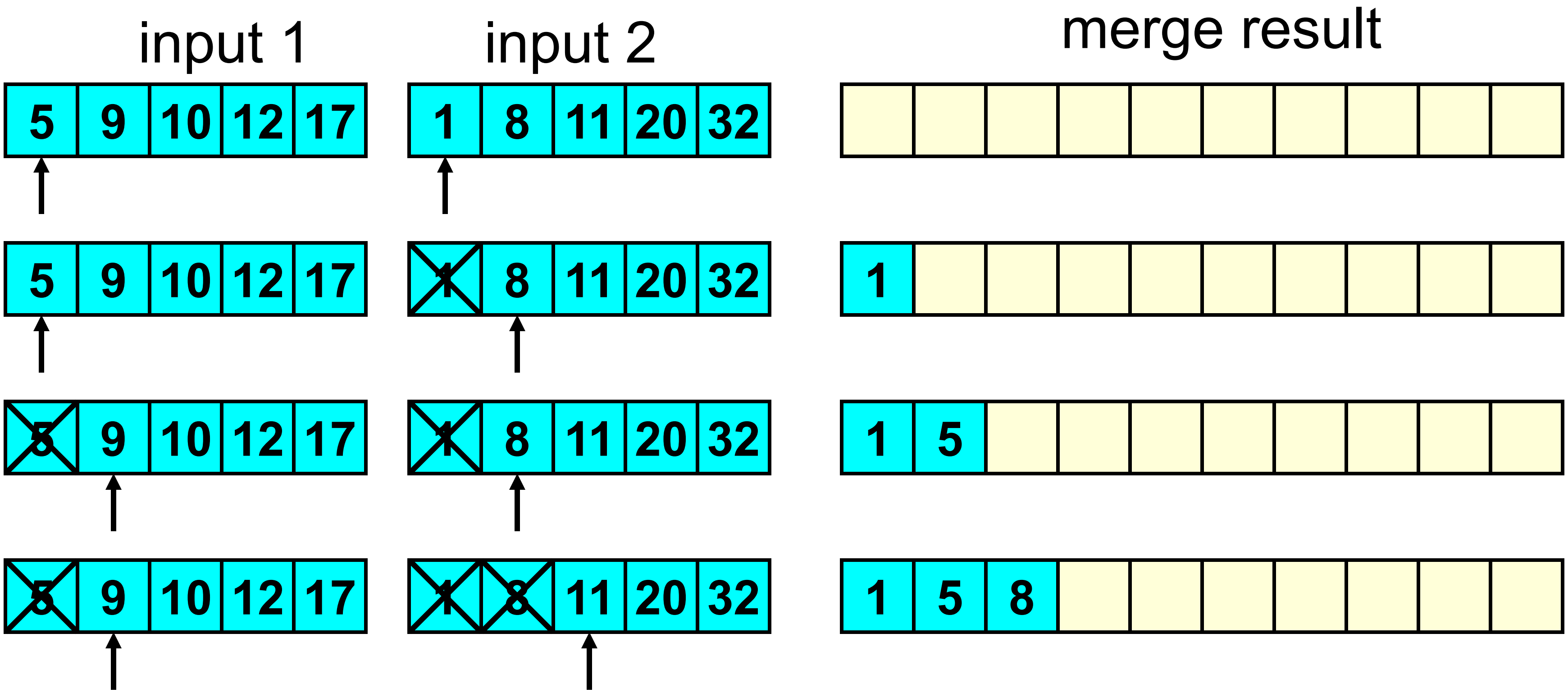
Fast Merge


input 1
input 2
merge result
5
9
10
12
17
1
8
11
20
32
5
9
10
12
17
1
8
11
20
32
1

Fast Merge

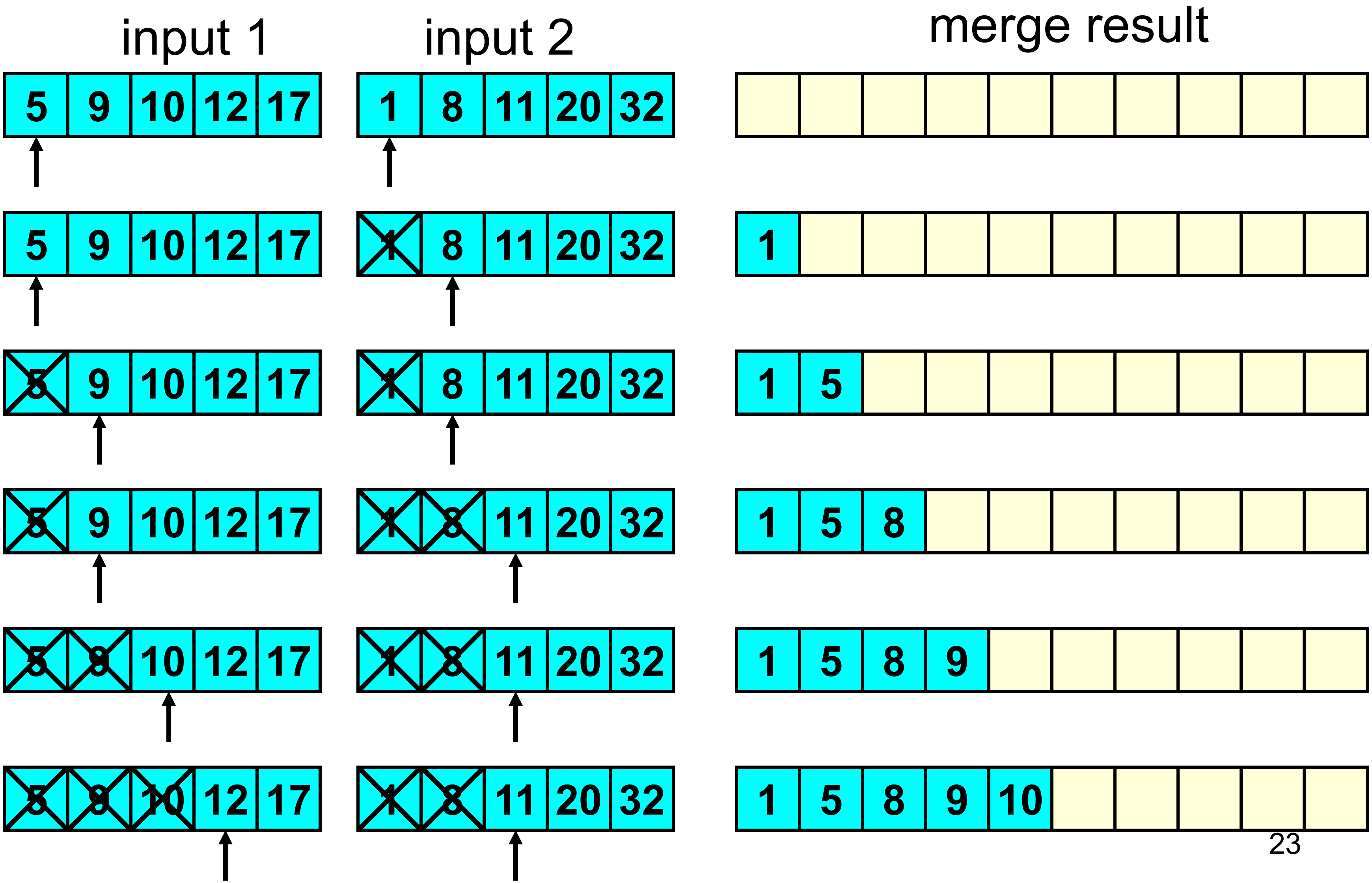


Fast Merge



input 1
input 2
merge result
5 9 10 12 17
1 8 11 20 32
5 9 10 12 17
1 8 11 20 32
1
5 9 10 12 17
1 8 11 20 32
1 5
5 9 10 12 17
1 8 11 20 32
1 5 8

Fast Merge



Complexity of Merge

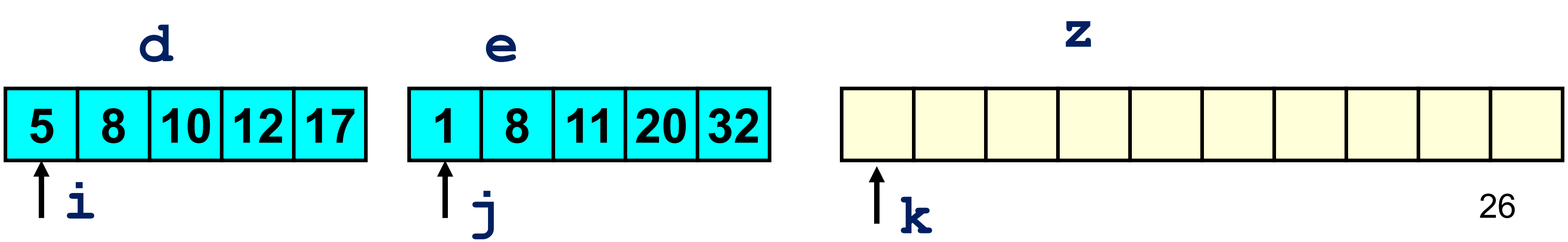
- What is O(??)

Set Operations

- Union is a special case of Merge
- Intersection is a special case of Merge
- Difference can be derived from Union and Intersection

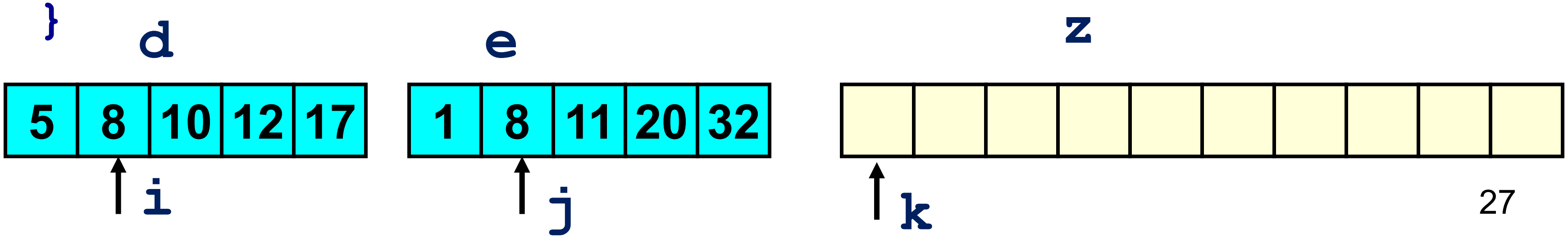
Intersection

```
/*i, j are indices of two bags  d, e
k is the index for intersection z*/
while (i < d_size && j < e_size){
   ...



}
```



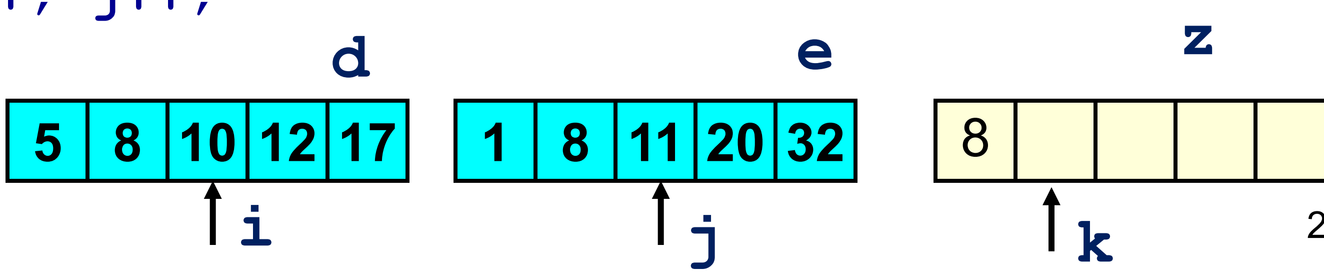
Intersection

```
/*i, j are indices of two bags  d, e
k is the index for intersection z*/
while (i < d_size && j < e_size){
  if (d[i] < e[j])
    i++;
  else if (d[i] > e[j])
         j++;
       else{ /*equal*/

         ...

       }
}
```



Intersection

```
/*i, j are indices of two bags  d, e
k is the index for intersection z*/
while (i < d_size && j < e_size){
  if (d[i] < e[j])
    i++;
  else if (d[i] > e[j])
         j++;
       else{ /*equal*/
         if(z[k] != d[i]){
            "add d[i] to z";
            k++;
          }
          i++; j++;
       }
}
```



Example: Intersection

```
/*i, j are indices of two bags  d, e
k is the index for intersection z*/
while (i < d_size && j < e_size){
  if (d[i] < e[j])
    i++;
  else if (d[i] > e[j])
         j++;
       else{ /*equal*/
         if(z[k] != d[i]){
            "add d[i] to z";
            k++;
          }
          i++; j++;
       }
}
```

Example: Union (unique elements)

```
/*i, j are indices of two collections  d, e
k is the index for union u*/

while (i < d->size && j < e->size){
   if (d[i] < e[j]){
      if(d[i] != u[k]){add d[i] to u; k++;}
      i++;}
   else if(e[j] < d[i]){
      if(e[j] != u[k]){add e[j] to u; k++;}
      j++;}
   else{ if(e[j] != u[k])
        {add e[j] to u; k++;}
        i++; j++;}
}
if (i == d->size){add rest of e to union}
if (j == e->size){add rest of d to union}
```

Difference (D - E)

`i, j` are indices of two collections **`d, e`**

```
while (i < d->size && j < e->size){
   if (d[i] < e[j]){add d[i] to diff; i++;}
   else if (d[i] > e[j]) j++;
        else {i++; j++;}
}
if (j == e->size && i < d->size){
   add rest of d to diff
}
```