ALGORITHMS: LINEAR SEARCH

ARRAY LINEAR SEARCH

- To find an item in an array, start at the beginning and check every item

```
public int search (int searchKey){

   for(int j=0; j<array.length; j++){
    if(array[j] == searchKey){
        return j;
    }
   }
   return null;
}
```

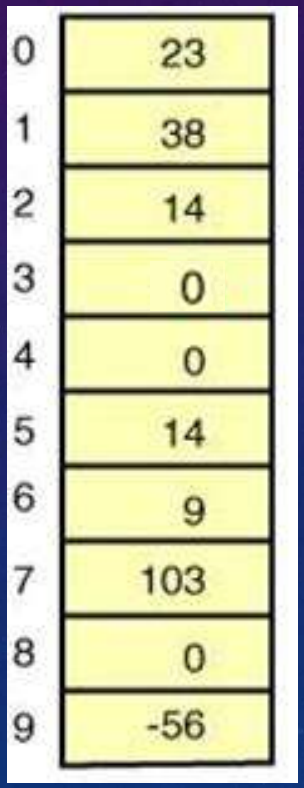
COUNTING MATCHES

- Count the number of items with a searchKey greater than a specified threshold

```
public int countMatches (int threshold){
   int count=0;
   for(j=0; j<array.length; j++){
    if(array[j] > threshold){
        count++;
    }
   }
   return count;
}
```

FINDING THE MAXIMUM OR MINIMUM

- Algorithm:
  - Initialize a candidate with the starting element
  - Compare candidate with remaining elements
  - Update it if you find a larger or smaller value



FIND BIGGEST

- Find the biggest value in the array
- Go through every element and track biggest found so far

```
public int findMax(){
   int biggestSoFar = array[0];
   for(int j=1; j<array.length; j++){
    if(array[j] > biggestSoFar){
        biggestSoFar=array[j];
    }
   }
   return biggestSoFar;
}
```

INSERTING INTO AN ARRAY

- Arrays have fixed size and may not be filled to capacity
- Some slots will be filled whereas others will be empty

[4 6 2 7 9 8 _ _ _ _ _]

- When a new element is added, it makes sense to add it to the next available free slot

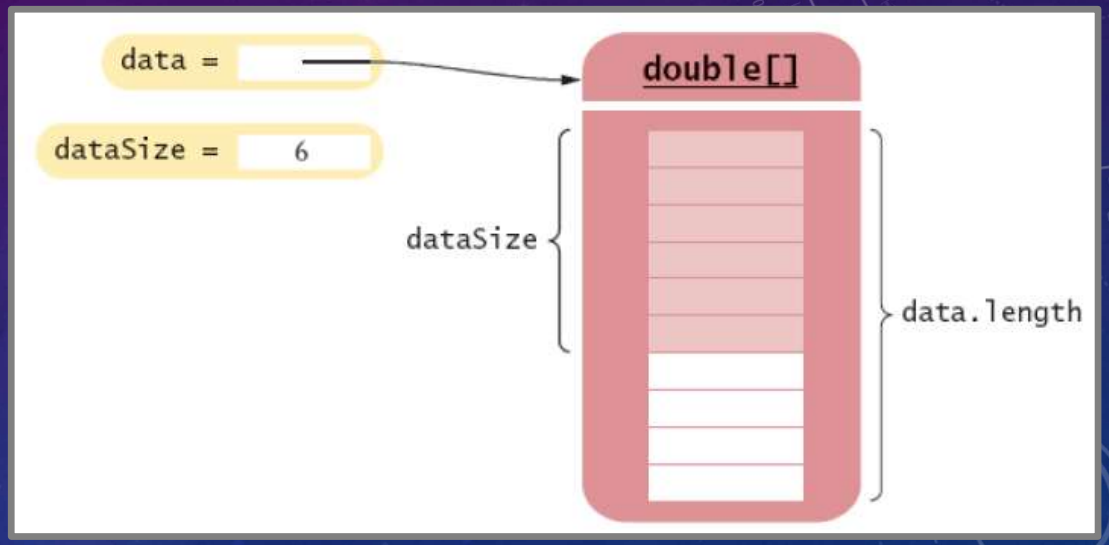
INSERTING INTO AN ARRAY

- If we know how many **elements** (not slots) are in the array then we know what the next available slot number is
- We can use a variable to track how many elements are currently in the array
- For example, we can create a variable called dataSize that we increase every time we add some data to our array.
- iIf dataSize = 6 this means there are six elements in the array and the next available slot will be the seventh slot, namely data[6]

```
final int LENGTH = 100;
double[] data = new double[LENGTH];
int dataSize = 0;
```

INSERTING INTO AN ARRAY

A Partially Filled Array



- Next element inserted goes in slot [`dataSize`]
- Update `dataSize` as array is filled:

```
public void insert (int value){
    data[dataSize] = value;
    dataSize++;
}
```

GROWING AN ARRAY

- If the array is full and you need more space, you can “grow” the array:
- (We are not really growing our array)
  - Create a new, larger array

```
double[] newData = new double[2 * data.length];
```

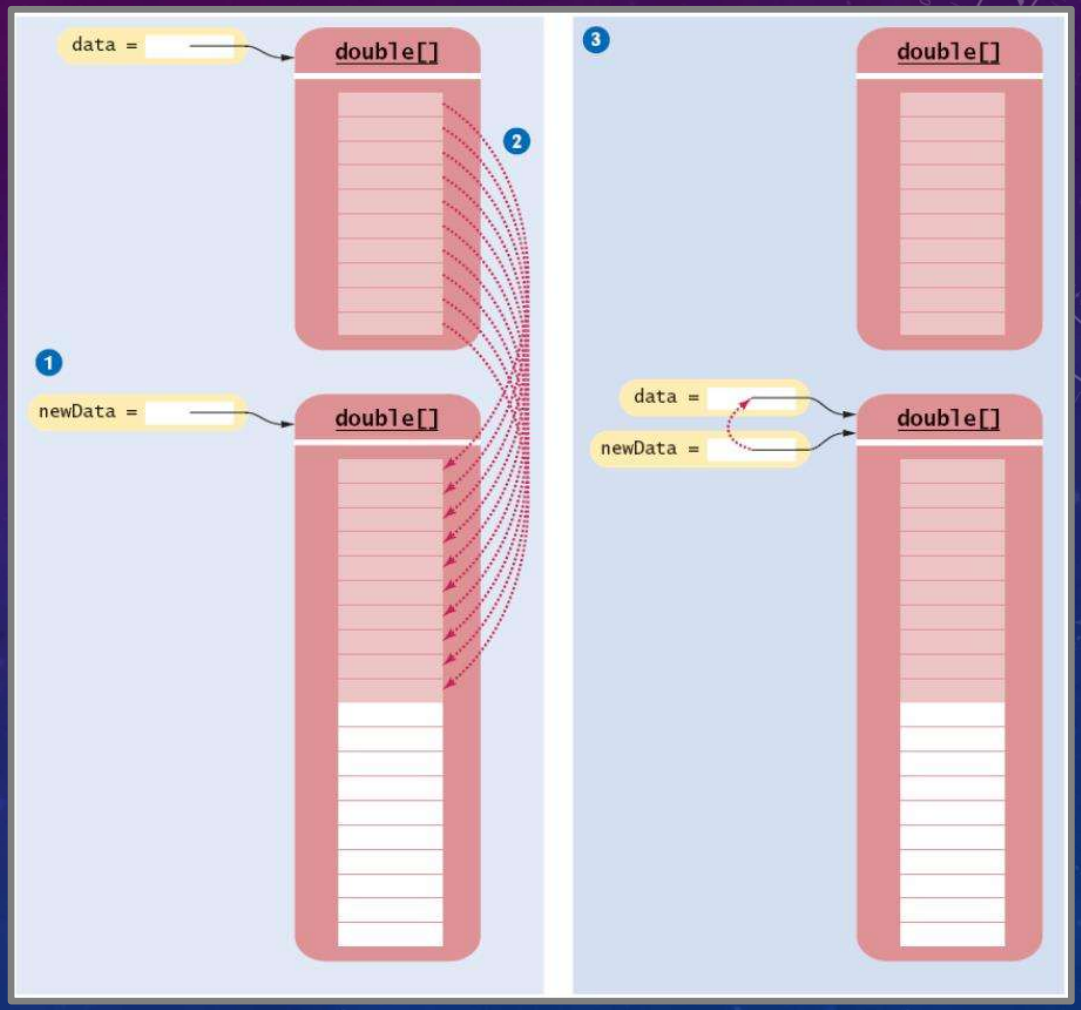
  - Copy all elements into the new array

```
for(int i=0;i<data.length;i++){
  newData[i]=data[i];
}
```

  - Change the reference to point to the new array

```
data = newData;
```

GROWING AN ARRAY



PROBLEMS WITH ARRAYS

You need to keep track of what slot in the array is the next free one

You need to write special code to search and delete a particular element

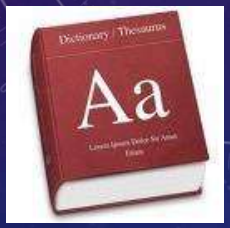
Every time you want to find an item, you have to check **EVERY** item

Every time you want to delete an item you have to check **EVERY** item

As the array gets bigger and bigger it will take longer and longer to find what you want

Imagine looking for a word in a dictionary and having to check every word!

ORDERED ARRAYS



- Checking every item in an array is known as *linear search*
- Do you preform a linear search when looking for a book in the library?
- Dictionaries and telephone directories are *ordered.*
- *Why?*
- It makes it easier to find stuff we're looking for
- If information is ordered then you can use a *binary search*

ORDERED ARRAYS

If an array is in order and we want to search for a particular entry, then we can play a guessing game

We try the middle element first (like guessing 50 for a number between 1 and 100)

If the middle element is smaller than the one we're looking for, we know that the element must be in the second half

If the middle element is bigger then the one we're searching for must be in the first half

GUESSING GAME

Computer: Guess my number between 1 and 100

You: 50

Computer: Too low!

You: Aha, the number must be between 50 and 100. I guess 75

Computer: Too high!

You: Aha, the number must be between 50 and 75. I guess 63 which is in the middle again.

Computer: Too low!

You: Now I'm getting close. The number must be between 63 and 75. How about 69?

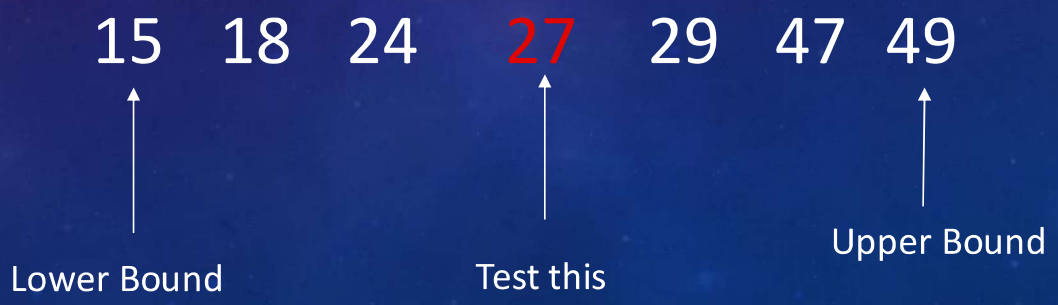
Computer: Too high!

You: 66

Computer: You got the correct answer! There were 100 numbers but you guessed in only 5 guesses!

WHAT DO WE NEED?

- We keep dividing our search space and therefore need to keep track of the limits
  - Upperbound
  - Lowerbound
- If the number is bigger than 27 then 27 is the new lower bound



ALGORITHM DESCRIPTION

- Objective : find the array slot in which a certain searchKey value is contained
- Identify the range of slots in which the value could possibly turn up
- Keep doing this while the search space is bigger than 0:
  - Check the slot in the middle of the range
  - If this slot holds the correct value, return that slot number
  - If not, adjust the search range and repeat the process

CODE

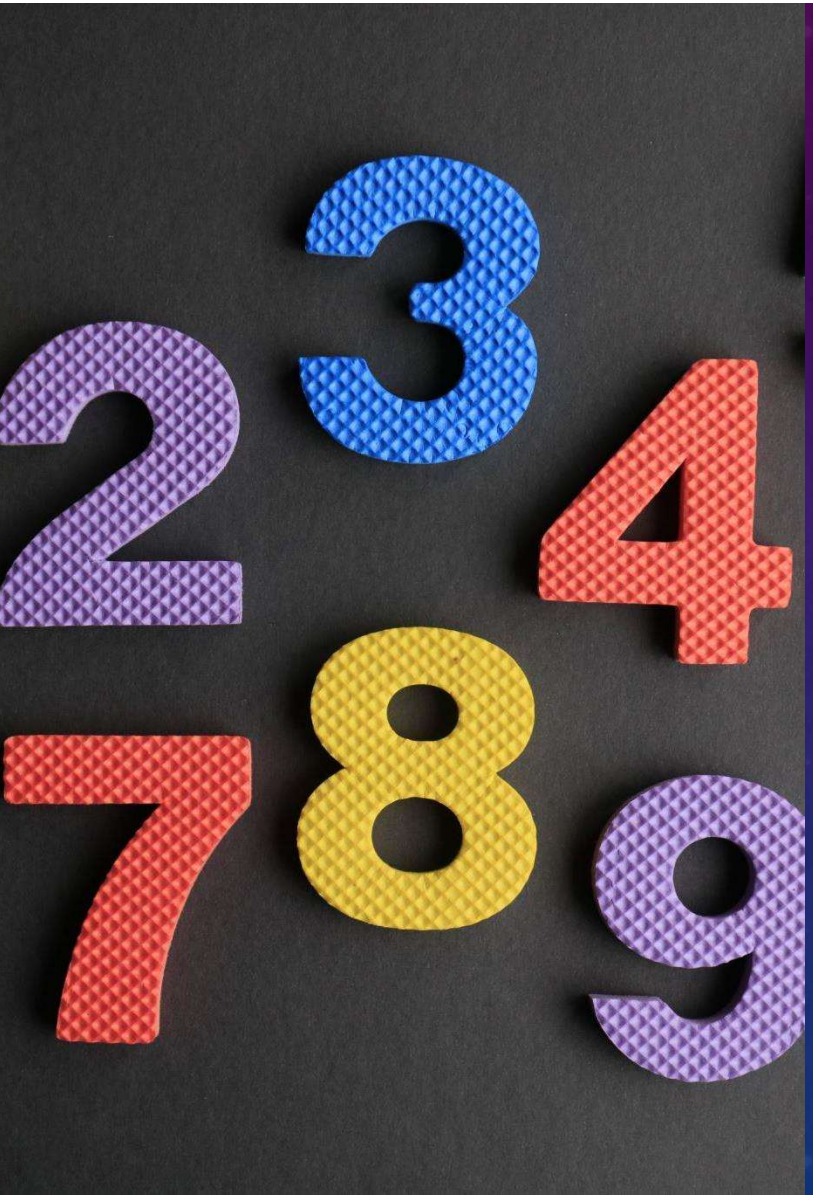
- In the following code we use the following variables:
  - `searchKey` is the number we're looking for
  - `nElems` is the number of elements in the array (it might not be full)
  - `lowerBound` and `upperBound` are used to track the range of our search
  - `check` is used to store the slot number we are currently checking
  - `myArray` is the array we're searching through

```
public int find(int searchKey){
     int lowerBound = 0;
     int upperBound = nElems-1;
     int check;

     while(true){
        check = (lowerBound + upperBound ) / 2;
        if(myArray[check]==searchKey){
           return check;              // found it
        }else if(lowerBound > upperBound){
           return -1;             // can't find it
        }else{                         // divide range

           if(myArray[check] < searchKey){
              lowerBound = check + 1; // it's in upper half
           }else{
              upperBound = check - 1; // it's in lower half
            }

       }  // end else divide range
    }  // end while
}  // end find()
```

Binary Search

KEEPING THINGS ORDERED

- In order to be able to run a binary search like this, the array we're working with has to be sorted
- So that creates a need for us – for our search algorithm to work for us, to we need algorithms to keep our array ordered
- Whenever a new number is inserted, it has to be inserted into the correct place
- Whenever a number is removed, the gap it leaves behind has to be filled



INSERTING AN ELEMENT

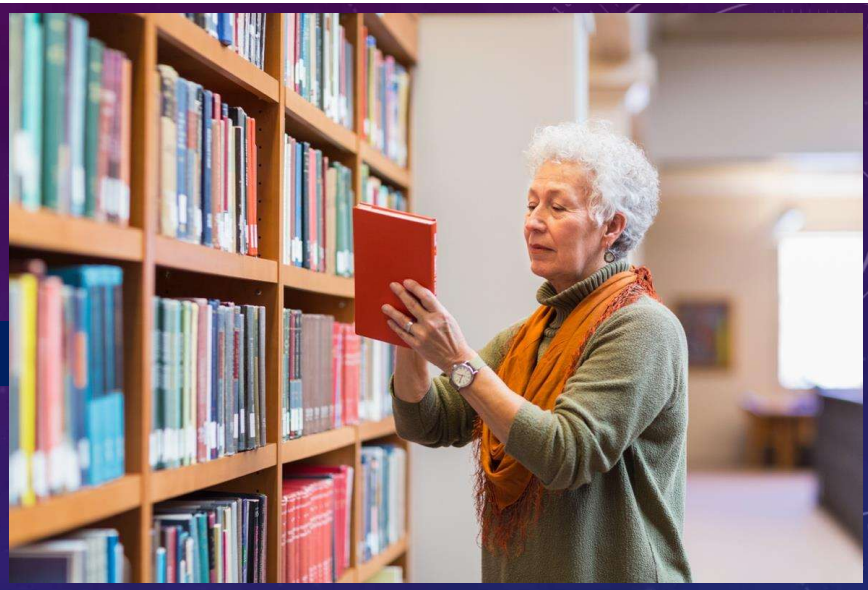
- We need to insert an element according to its order
- This means we will have to move all the other elements up to make room

[2 4 6 7 8 9 _ _ _ _ _]

- Say we want to insert the number 5
- This should go in the third slot (between 4 and 6)
- We need an algorithm that is going to shuffle all the elements from slot 2 onwards one space to leave a gap

INSERTING AN ELEMENT

Make a gap in the array by shifting everything up



2 4 6 7 8 9 _ _ _ _

```
//lets make space to insert something into slot 2

for(int j=dataSize;j>2;j--){
  data[j]=data[j-1];
}
```

2 4 _ 6 7 8 9 _ _ _

FULL INSERTION METHOD

```
public void insert(int value){

     int j=0;
    while(array[j] < value &&j<nElems){   //find where it goes
    j++;                                  //linear search
     }
    for(int k=nElems; k>j; k--){     // move bigger ones up
        a[k] = a[k-1];
     }
     a[j] = value;                          // insert it
     nElems++;                              // increment size
  }                  // end insert()
```

REMOVING AN ELEMENT

- Say we want to remove a particular element in our array
- Once we delete it there will be a gap left in our array
- If we don’t keep track of these gaps, then the array will just fill up with holes like a bookshelf with empty, used spaces
- We need an algorithm that will move all the elements down to fill the gap that is created after one is removed

REMOVING AN ELEMENT

Removing – remove an empty space by shifting everything down

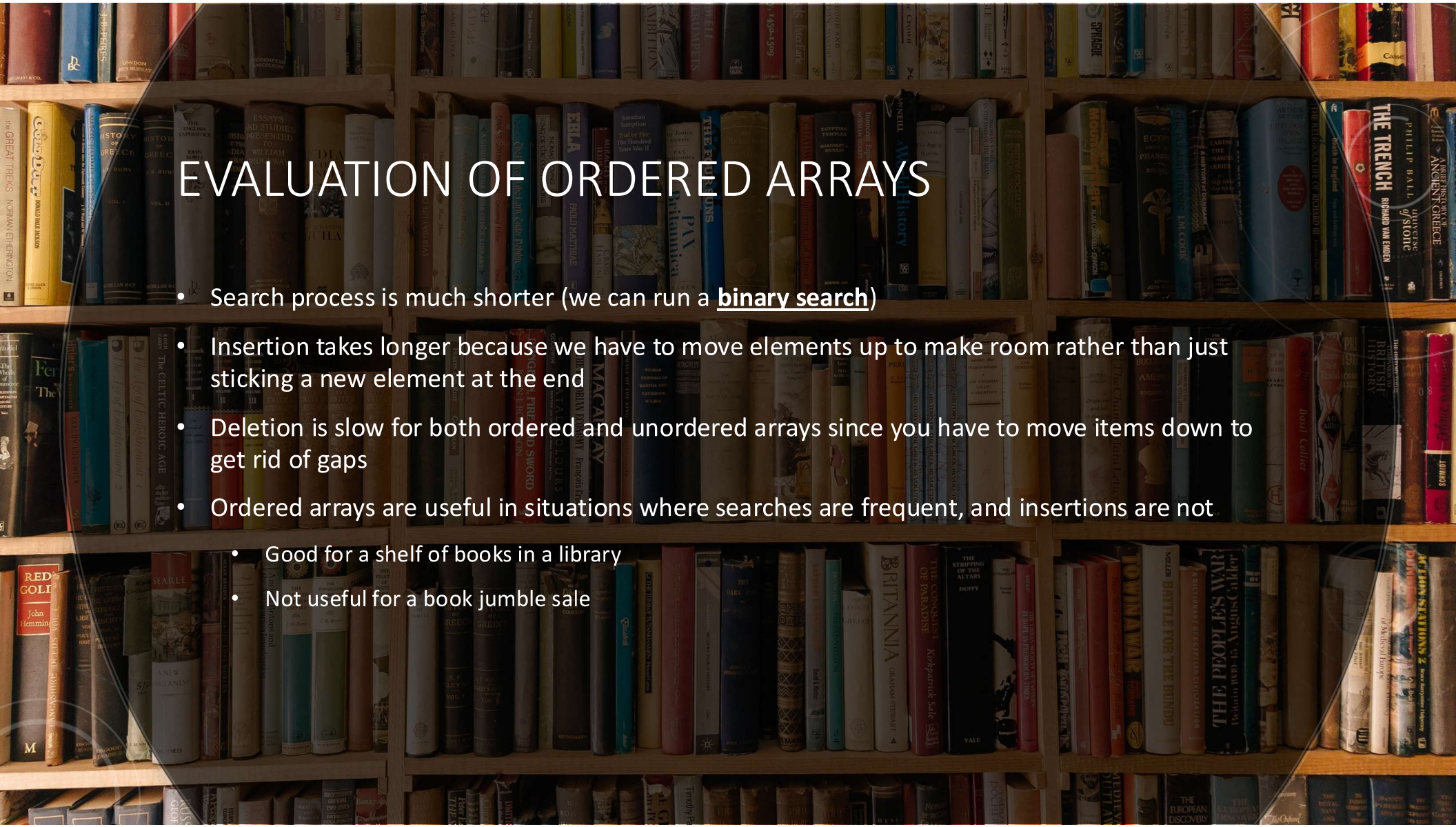
2 4 6 7 8 9 _ _ _ _

```
//delete something from slot 2

for(int j=2;j<dataSize;j++){
  data[j]=data[j+1];
}
```

2 4 7 8 9 9 _ _ _ _

REMOVING AN ELEMENT

```
public boolean delete(int value){

      int j=0;
      while(array[j] != value&&j<nElems){
   j++;
      }                    // linear search
      if(j==nElems){                        // can't find it
         return false;
      }else{                                   // found it

         for(int k=j; k<nElems-1; k++){  // move values down
            array[k] = array[k+1];
        }
         nElems--;                             // decrement size
         return true;
      }
}                       // end delete()
```



EVALUATION OF ORDERED ARRAYS

- Search process is much shorter (we can run a **binary search**)
- Insertion takes longer because we have to move elements up to make room rather than just sticking a new element at the end
- Deletion is slow for both ordered and unordered arrays since you have to move items down to get rid of gaps
- Ordered arrays are useful in situations where searches are frequent, and insertions are not
  - Good for a shelf of books in a library
  - Not useful for a book jumble sale

HOW GOOD IS BINARY SEARCH?

- As arrays get bigger, using a binary search becomes more important
- A linear search would take ages!

Size of Array	Comparisons Needed
10	4
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20
10,000,000	24
100,000,000	27
1,000,000,000	30

MATHEMATICALLY

- The number of steps needed to perform a binary search on an array of size N is the number of times that N can be halved

- If N is 16 then 4 steps will be needed
  - Step 1: narrow search space down to 8 slots
  - Step 2: narrow search space down to 4 slots
  - Step 3: narrow search space down to 2 slots
  - Step 4: narrow search space down to 1 slot

- Remember – we might get lucky, but we can't count on it!

- Each iteration of the binary search algorithm halves the search space that needs to be considered

- In other words, each extra iteration allowed doubles the range you can search through

A LOG RELATIONSHIP

- Each step halves the size, so the number of iterations needed to search through an array using a binary search is the number of times the size of the array can be halved

  $size = 2^{iterations}$

- The opposite of raising something to a power is to take its log

$iterations = \log_2(size)$

- Number of steps required increases very slowly compared to increases in size – logarithmically as opposed to linearly
- We express this log type relationship between array size and number of steps required by saying that the complexity of binary search is O(log n)