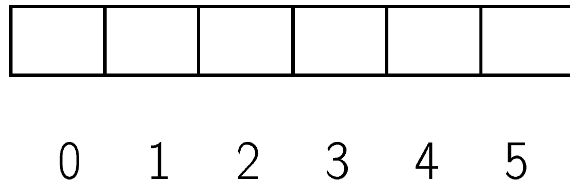
Arrays

- An *array* is an indexed sequence of variables of the same type. By indexed we mean that the variables are consecutive in memory and each of them has an index, with 0 being the first, 1 the second, and so on.



0 1 2 3 4 5

- Each variable in the array is called a *position*, a *cell* or a *slot*, and as any variable, it can contain a value.

Arrays

- An array is an ordered/indexed sequence of elements of the same type.
- Array declaration

```
type[] variable;
```

- Array creation:

```
variable = new type[integer-expression];
```

- Array reading access:

```
variable[integer-expression]
```

- Array writing access:

```
variable[integer-expression] = expression;
```

Initializing arrays

- If we have a class

```
class B
{
  int n;
  B(int x) { n = x; }
}
```

- and somewhere else we declare and create an array

```
B[] list = new B[7];
```

- Then all the slots in the array will be initialized to `null`. This is, the constructor for B will not be called. If we want an object created in each slot, we have to do it explicitely:

```
for (int i = 0; i < list.length; i++)
{
  list[i] = new B(3);
}
```

Array operations

- Adding elements
- Removing/deleting elements
- Finding elements
- Increasing the size of an array

Array operations

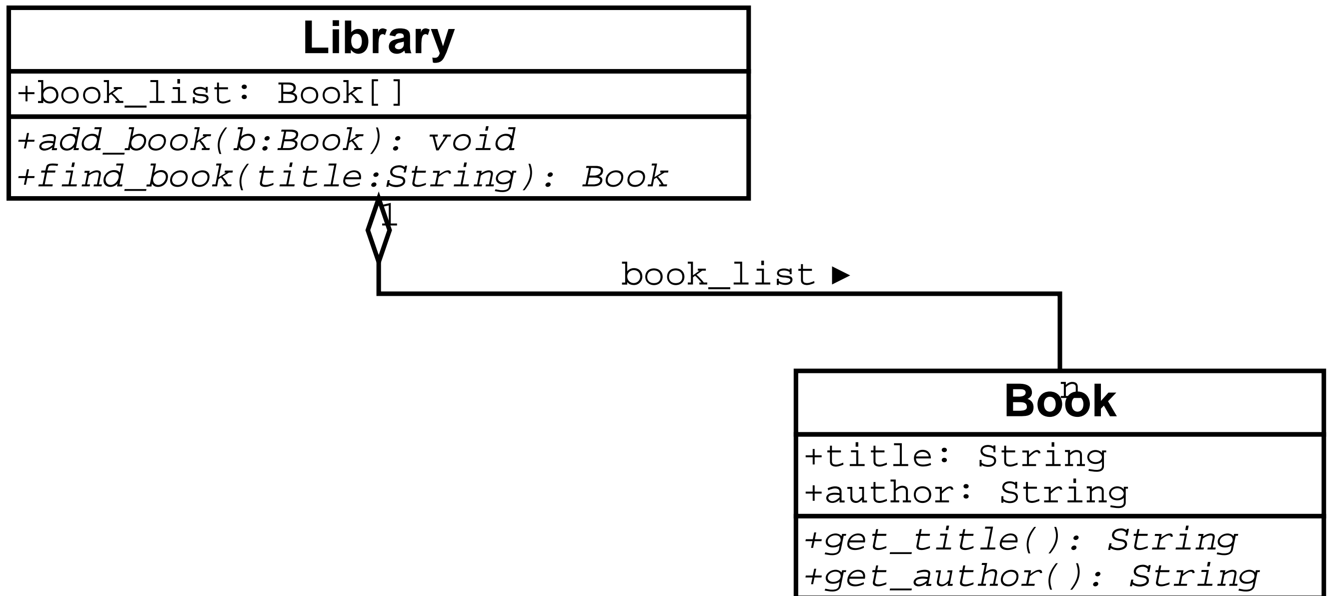
- Library: Book database

- Problem: Create a database of books, which supports the operations of adding a new book, and searching for a book by title.

- Analysis:
  - Identify objects and classes:
    * Individual books
    * A library: book database
  - Relationships
    * Each book *has a* title and an author
    * A book database *has a* list of books
  - Operations/Interactions/Behaviour
    * Adding books to a database
    * Searching for a book in a database

Array operations

- Design
  - Class diagram



Array operations

```
class Book
{
  private String title, author;
  public Book(String t, String d)
  {
    title = t;
    author = d;
  }
  public String title() { return title; }
  public String author() { return author; }

}
```

Array operations

```
class Library
{
  private Book[] book_list;
  public int next_available;

  public Library(int max_capacity) { ... }

  public int number_of_books() { ... } // Accessor

  public void add_book(Book m) { ... } // Mutator

  private void grow_array(int n) { ... } // Mutato

  public int book_index(String title) { ... } // A

  public Book find_book(String title) { ... } // A

  public void delete_book(String title) { ... } //
} // End of Library
```

McGill

Array operations

```
class Library
{
  private Book[] book_list;
  public int next_available;

  public Library(int max_capacity)
  {
    book_list = new Book[max_capacity];
    next_available = 0;
  }

  public int number_of_books()
  {
    return next_available;
  }

  // Continues below...
```

Array operations

```
public void add_book(Book m)
{
  // If available slot found, store it
  if (next_available < book_list.length)
  {
    book_list[next_available] = m;
  }
  // Otherwise
  else
  {
    int l = book_list.length;
    grow_array( (int)(l * 0.10) + 10 );
    book_list[l] = m;
  }
  next_available++;
}
```

Array operations

```
private void grow_array(int n)
{
  int new_capacity = book_list.length + n;
  Book[] new_list = new Book[new_capacity];
  int i = 0;
  while (i < book_list.length)
  {
    new_list[i] = book_list[i];
    i++;
  }
  book_list = new_list; // Update list reference
}
```

Array operations

```
public int book_index(String title)
{
  int i;
  i = 0;
  while (i < book_list.length)
  {
    Book m = book_list[i];
    if (m != null)
    {
      String s = m.title();
      if (s.equals(title))
      {
        return i;
      }
    }
    i++;
  }
  return -1;
}
```

Array operations

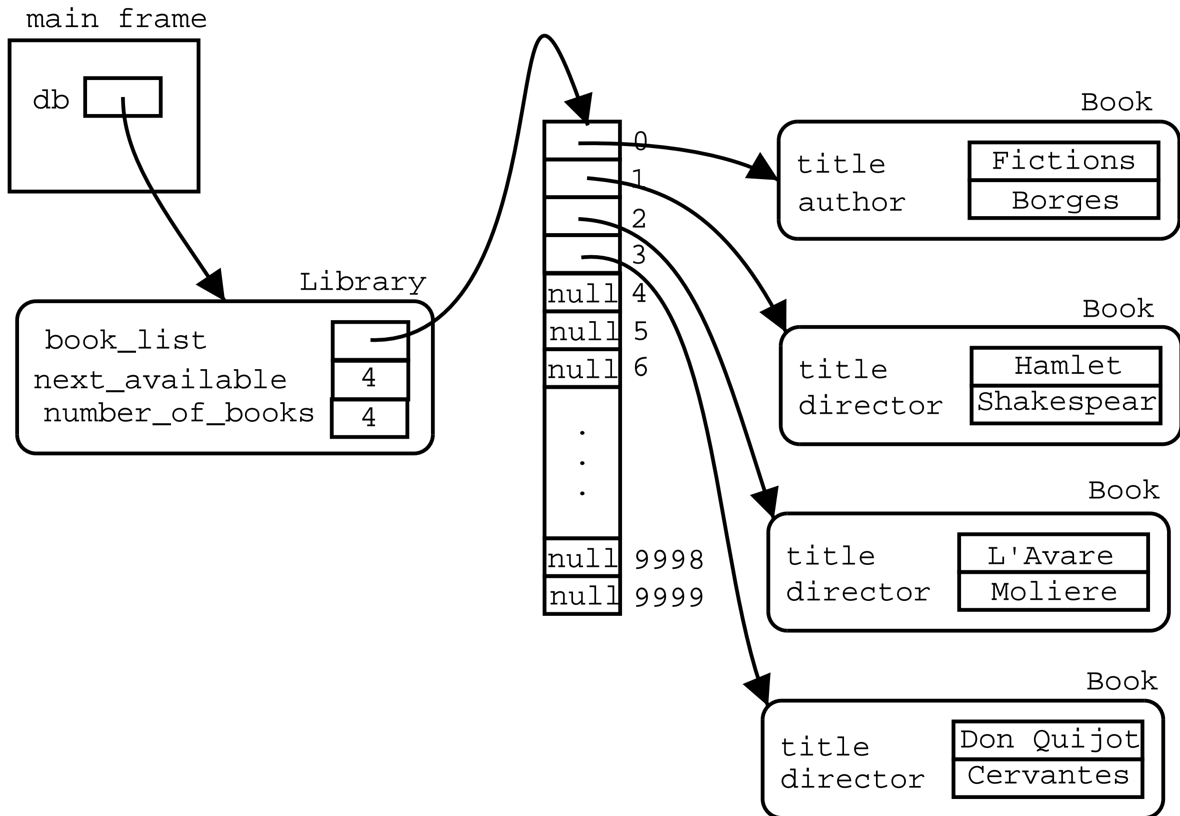
```
public Book find_book(String title)
{
  int i = book_index(title);
  if (i != -1) return book_list[i];
  return null;
}

public void delete_book(String title)
{
  int i = book_index(title);
  if (i != -1)
  {
    book_list[i]=book_list[next_available-1];
    book_list[next_available - 1] = null;
    next_available--;
  }
}
} // End of Library
```

Array operations

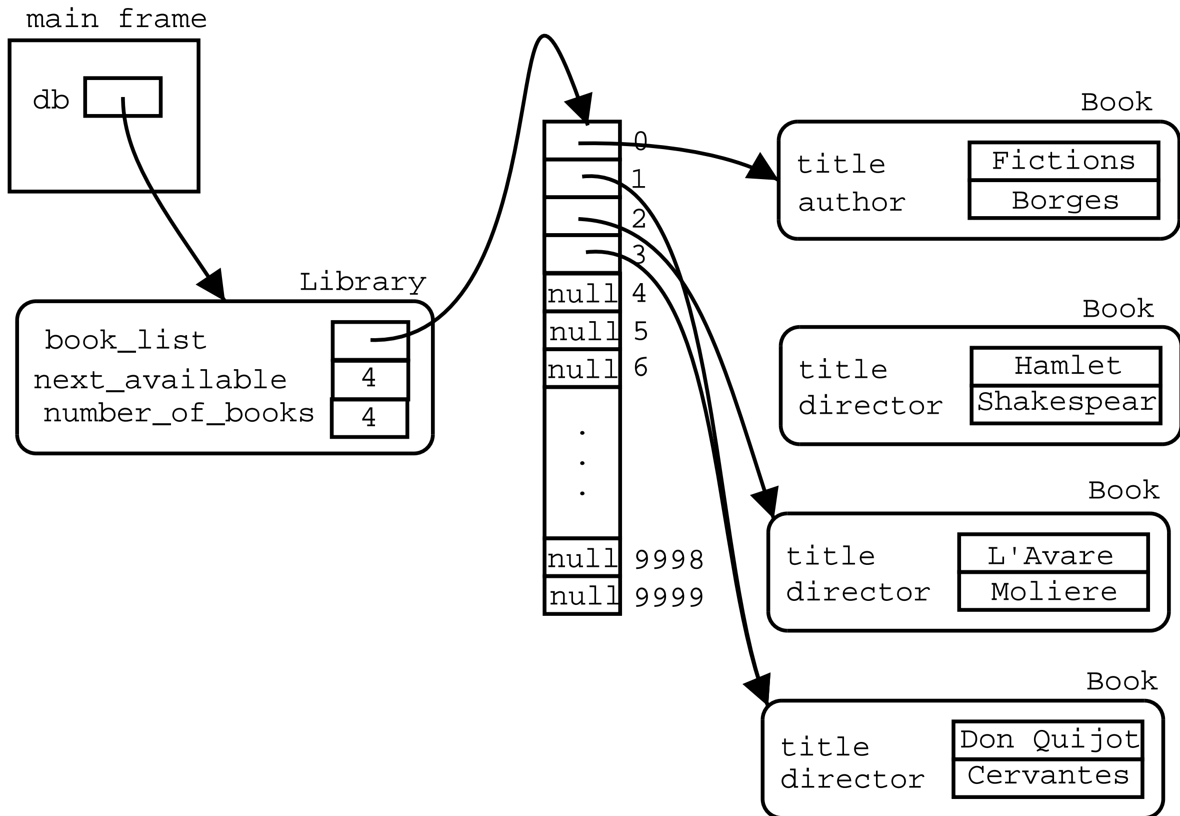
```
public class Test {
  public static void main(String[] args)
  {
    Library db = new Library(10000);
    Book m;
    m = new Book(``Fictions'',``Borges'');
    db.add_book(m);
    m = new Book(``Hamlet'',``Shakespeare'');
    db.add_book(m);
    m = new Book(``L\'Avare'',``Moliere'');
    db.add_book(m);
    db.delete_book(``Hamlet'');
    m = new Book(``Don Quijote'',``Cervantes'');
    db.add_book(m);
  }
}
```

Array operations

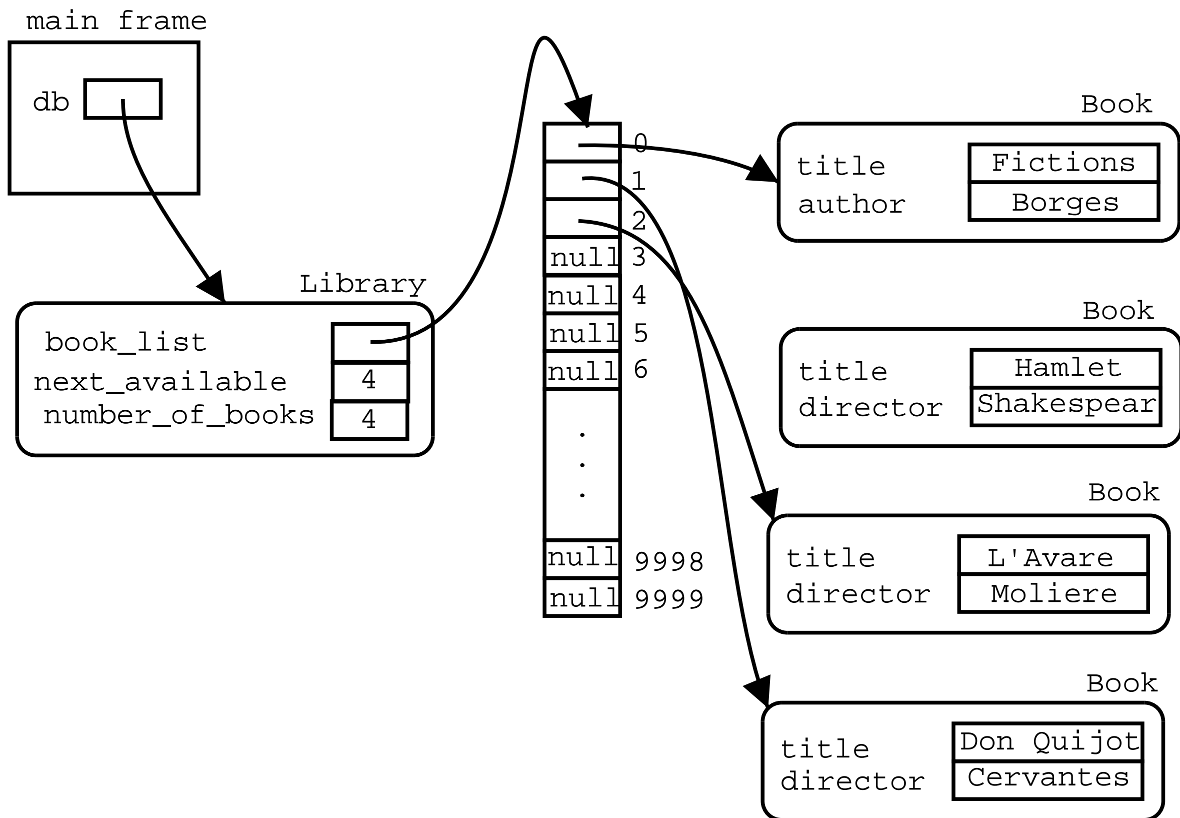


McGill

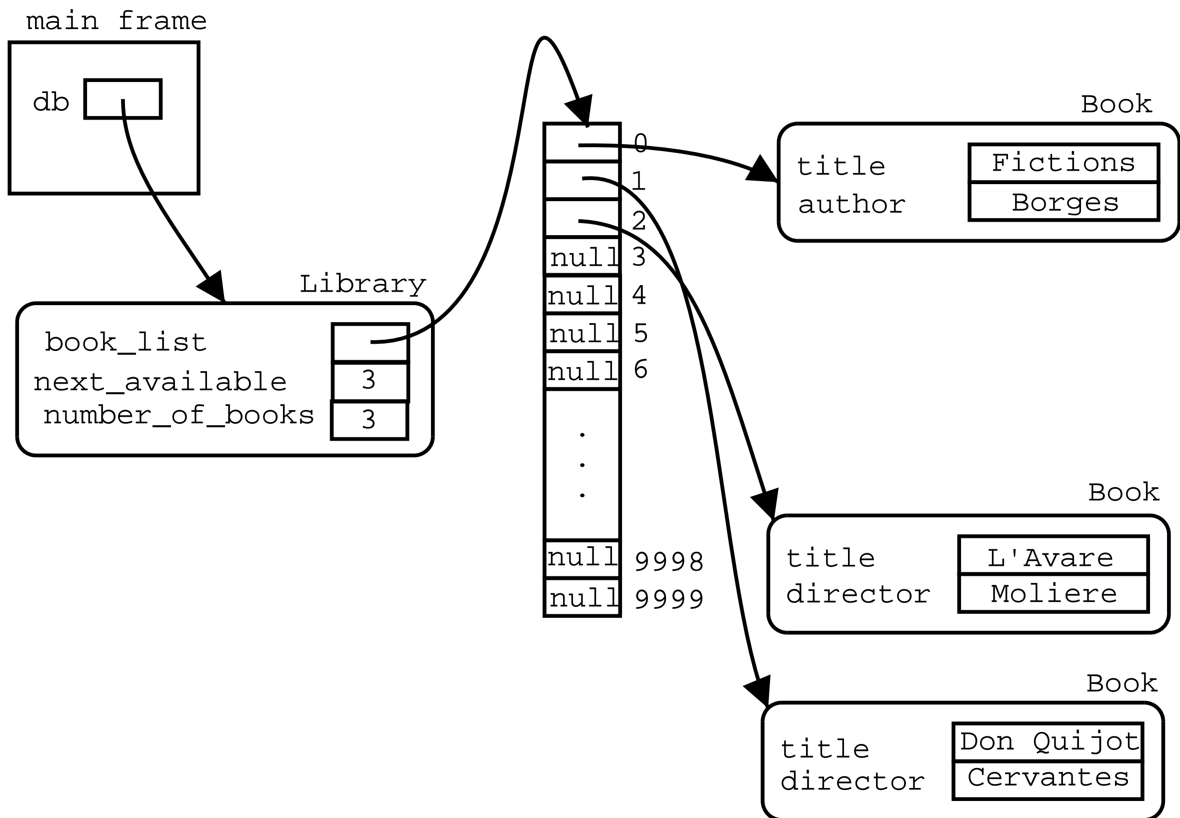
Array operations



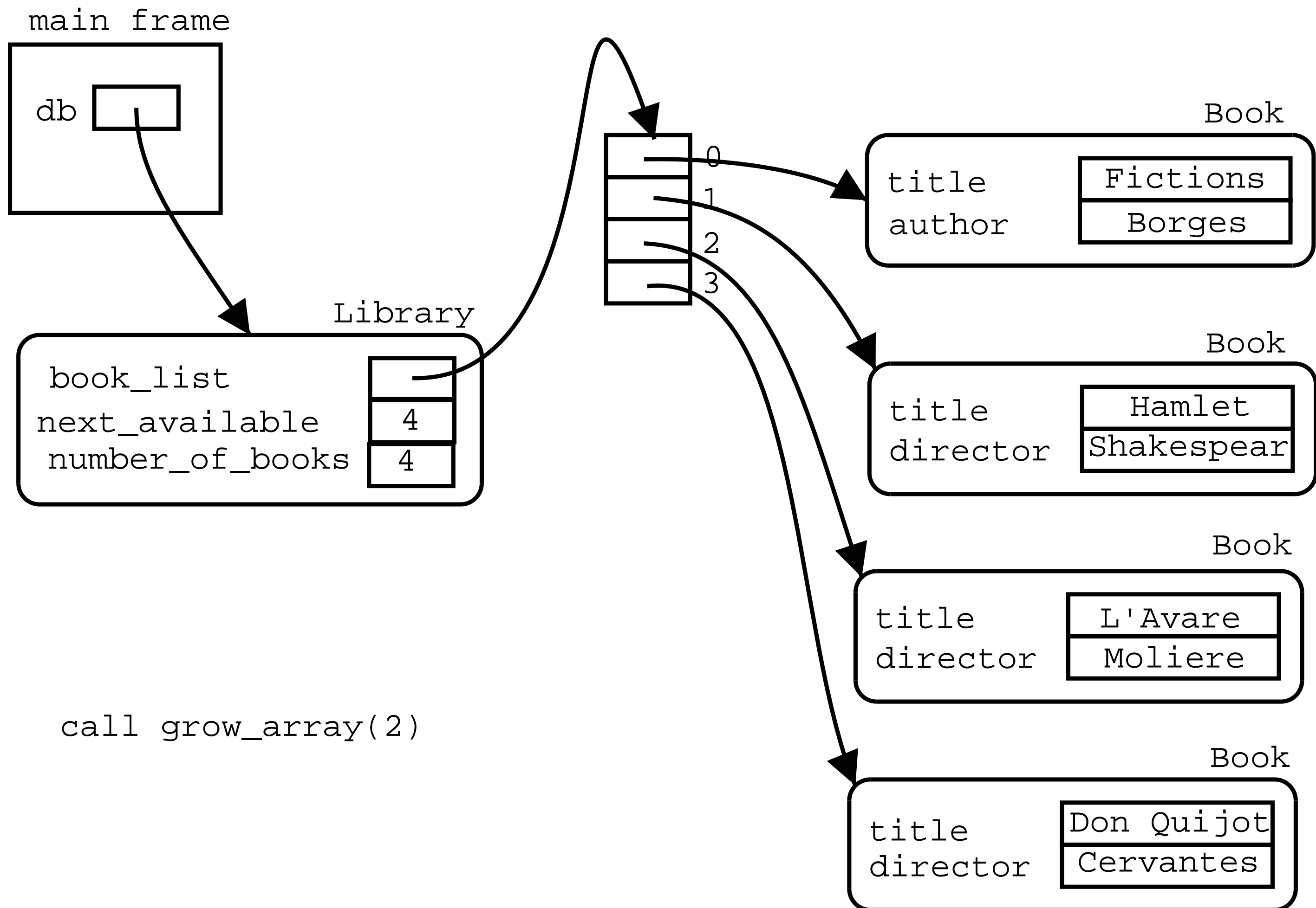
Array operations



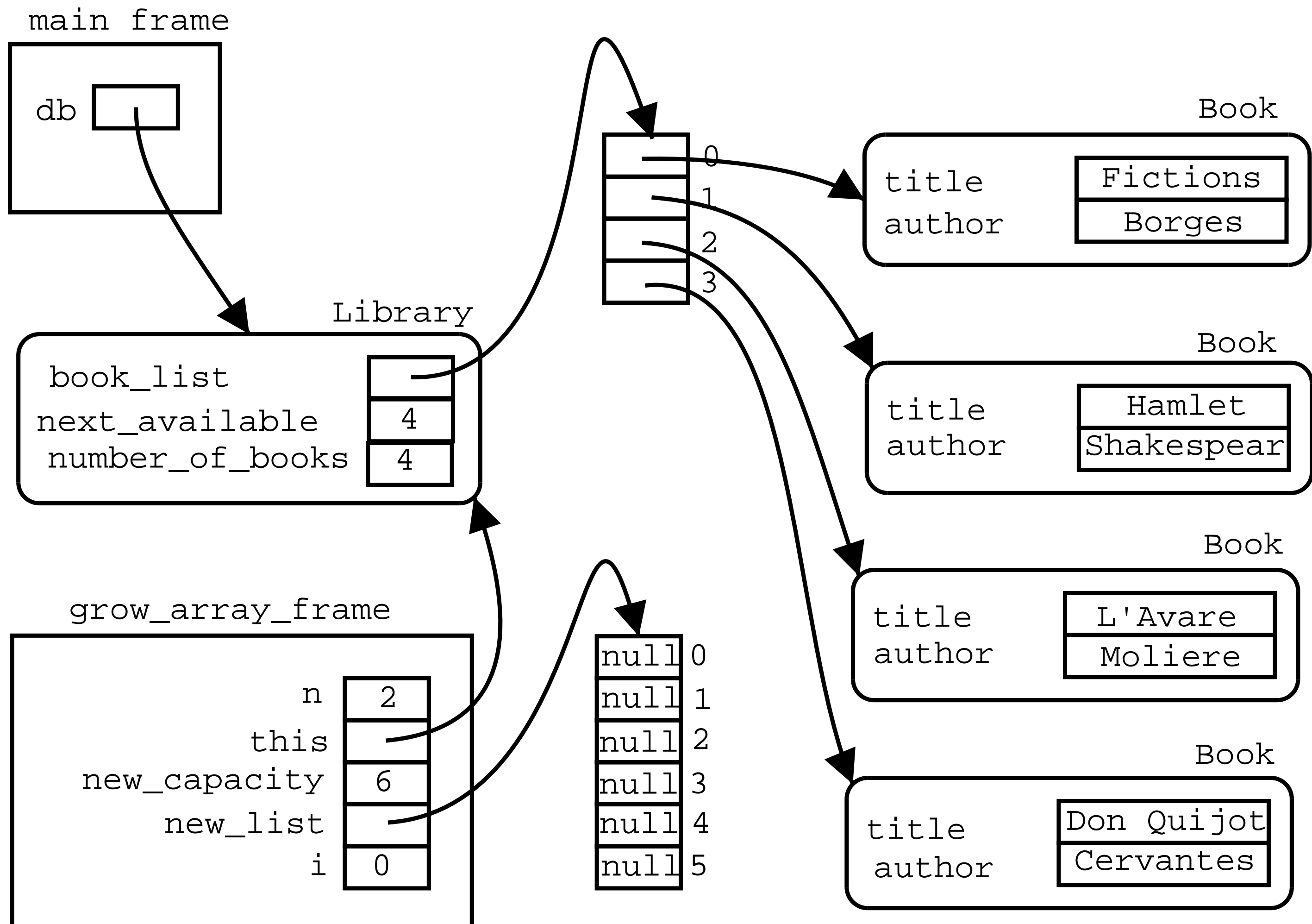
Array operations



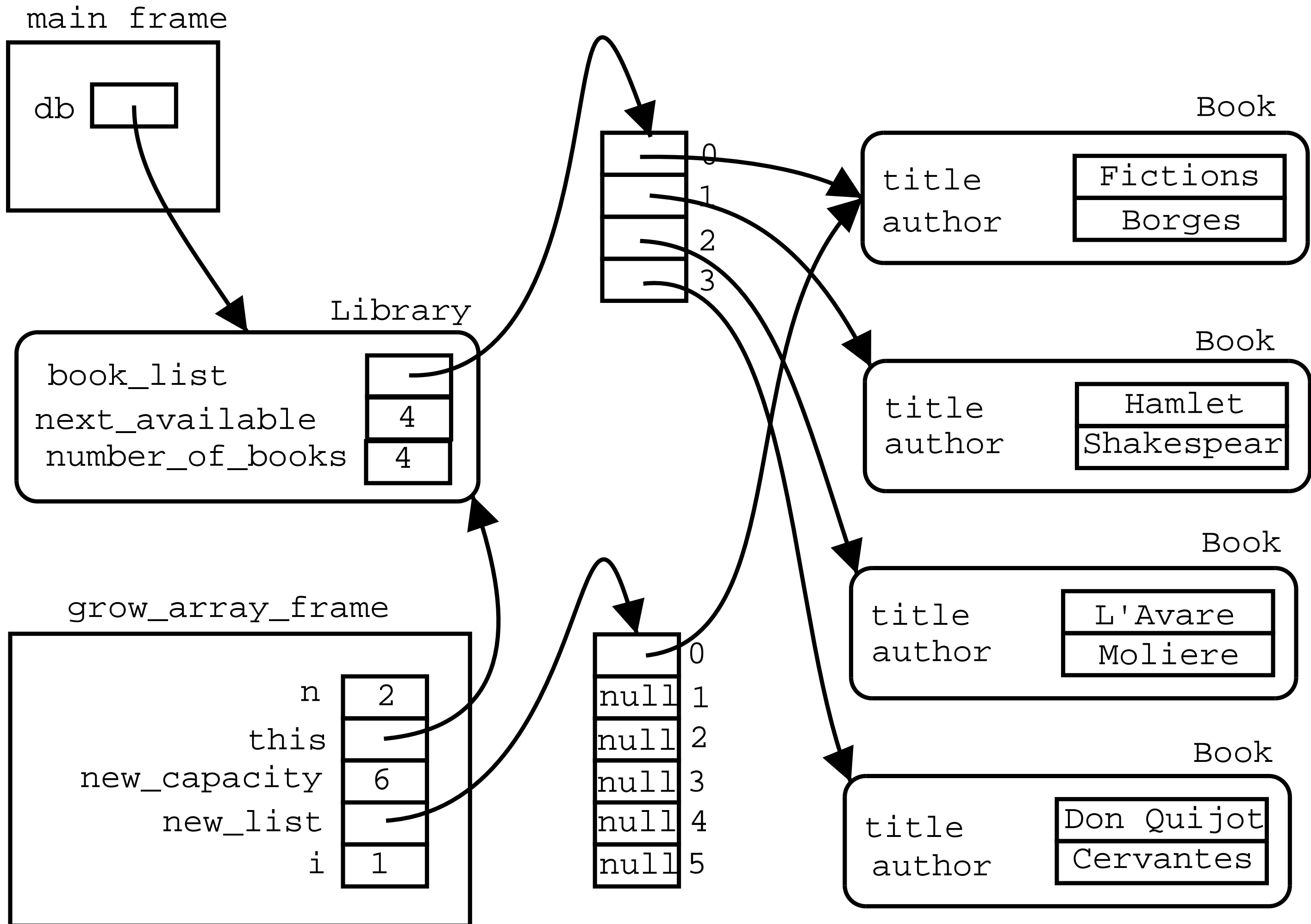
Growing arrays



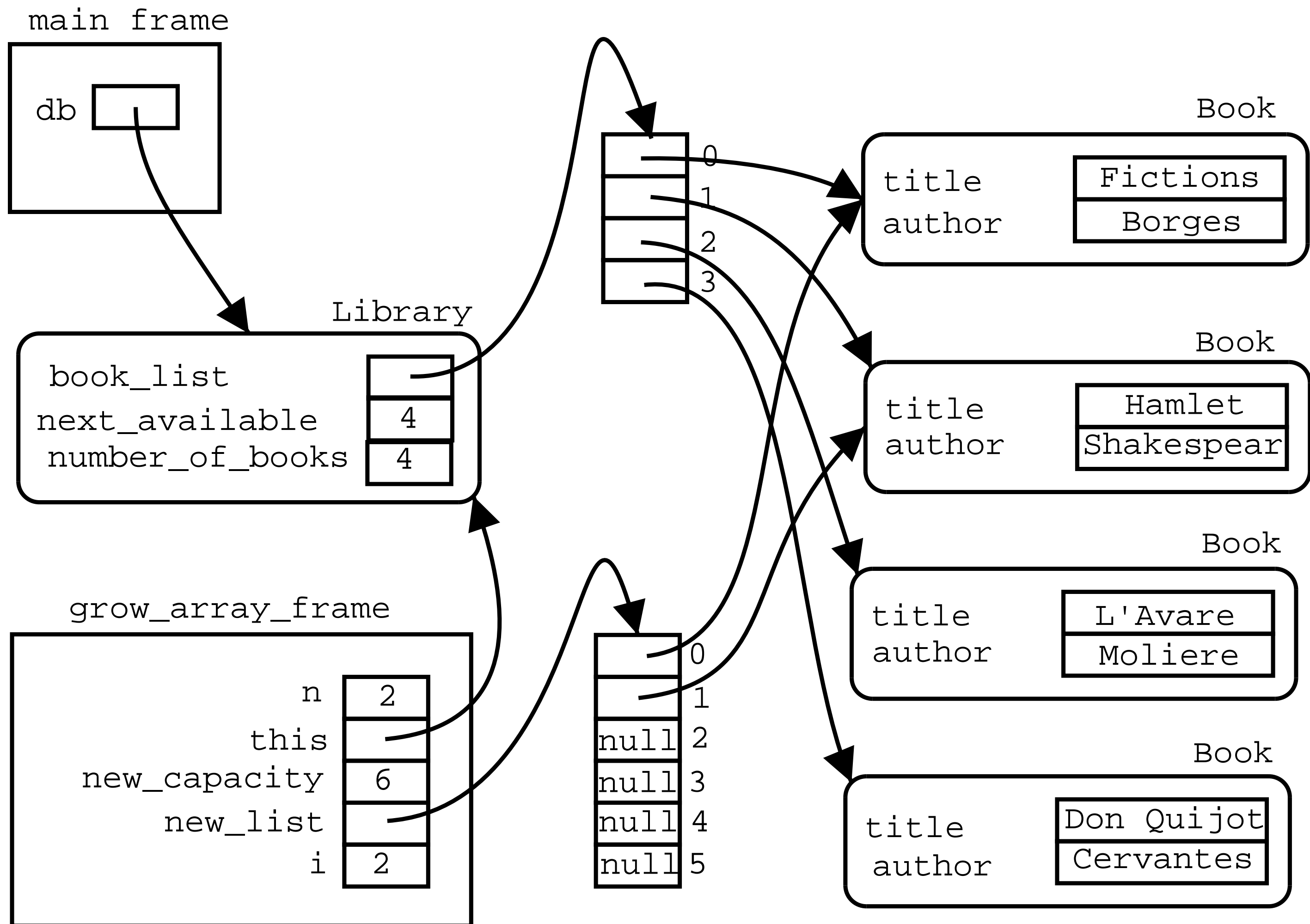
Growing arrays



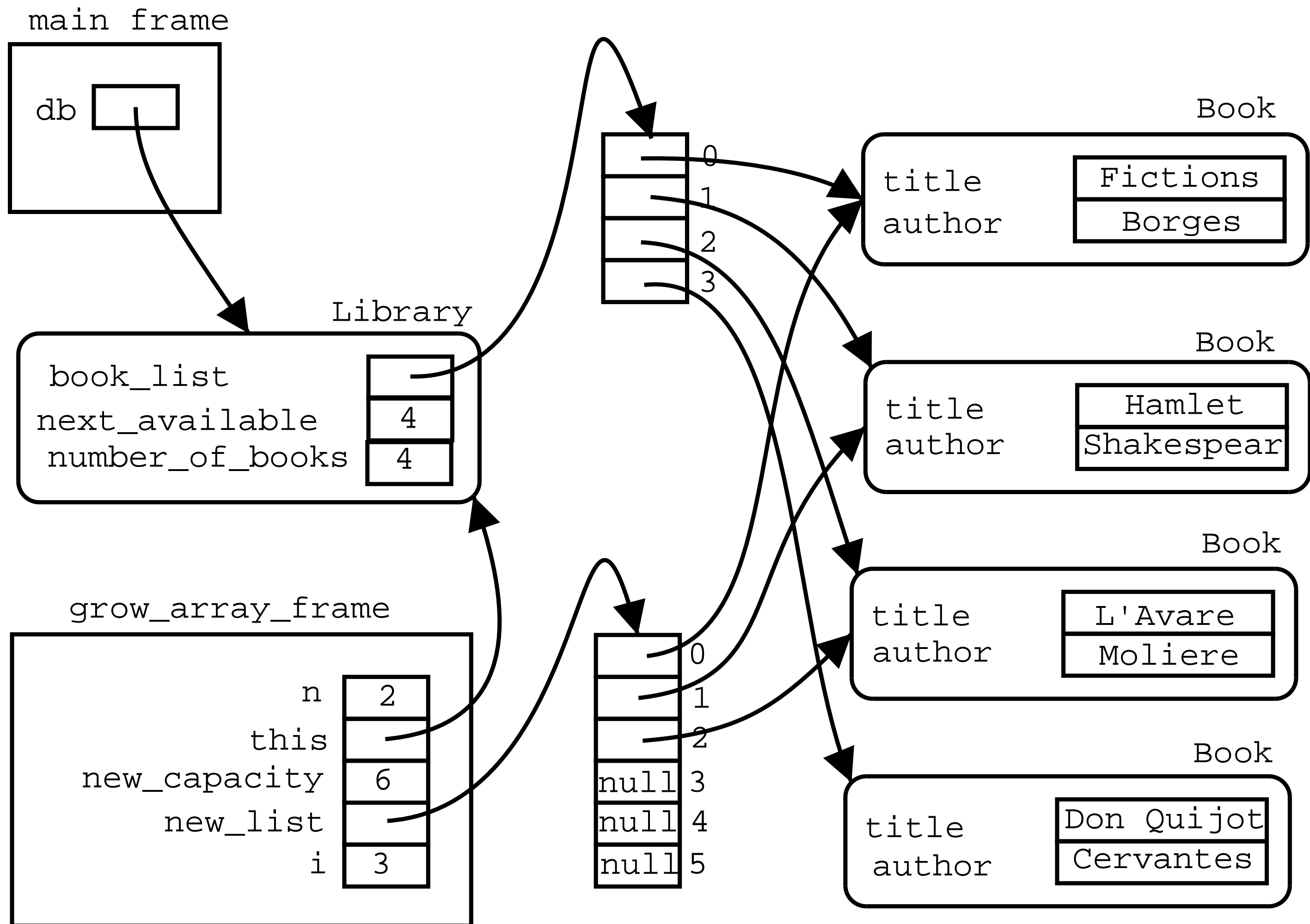
Growing arrays



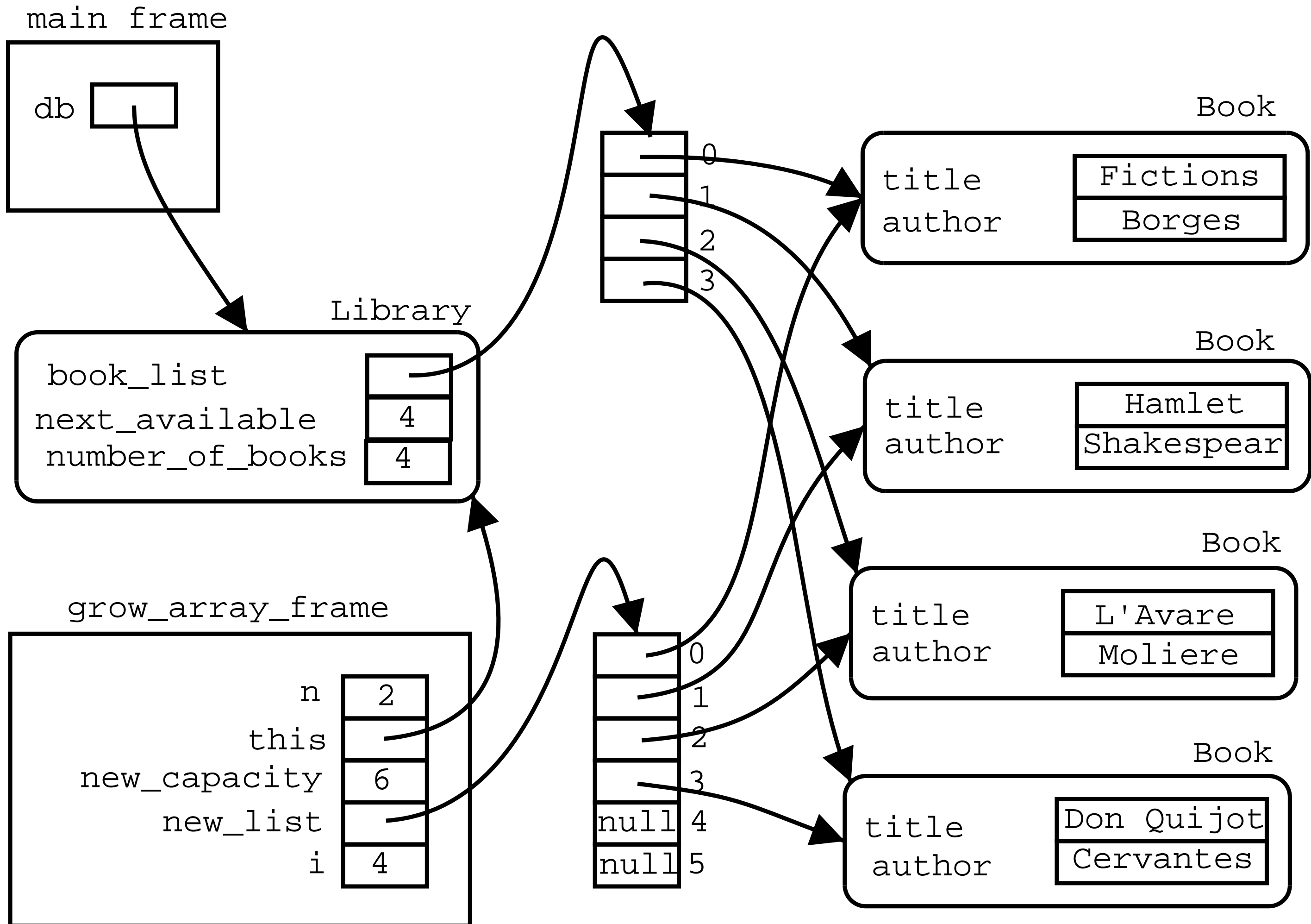
Growing arrays



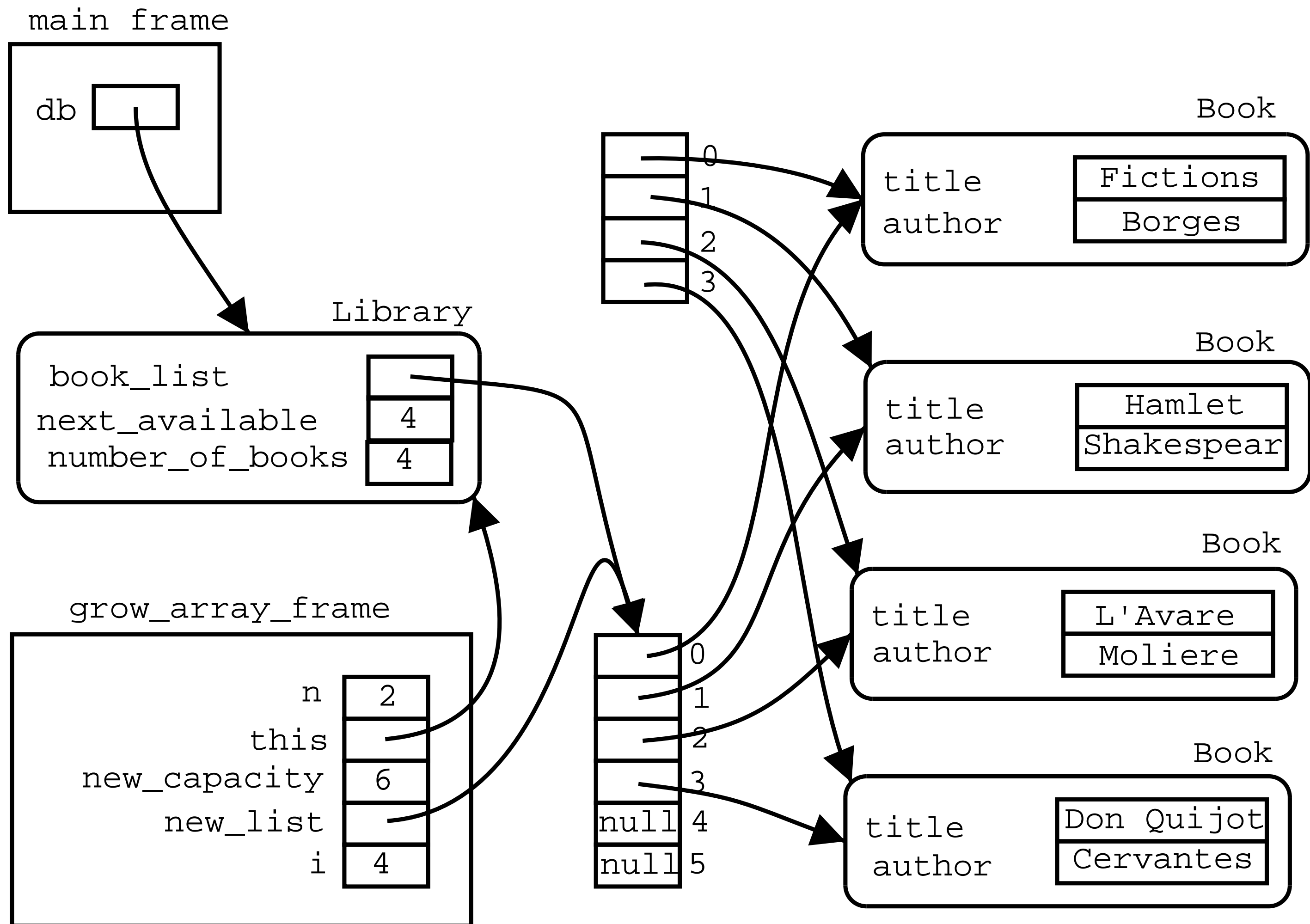
Growing arrays



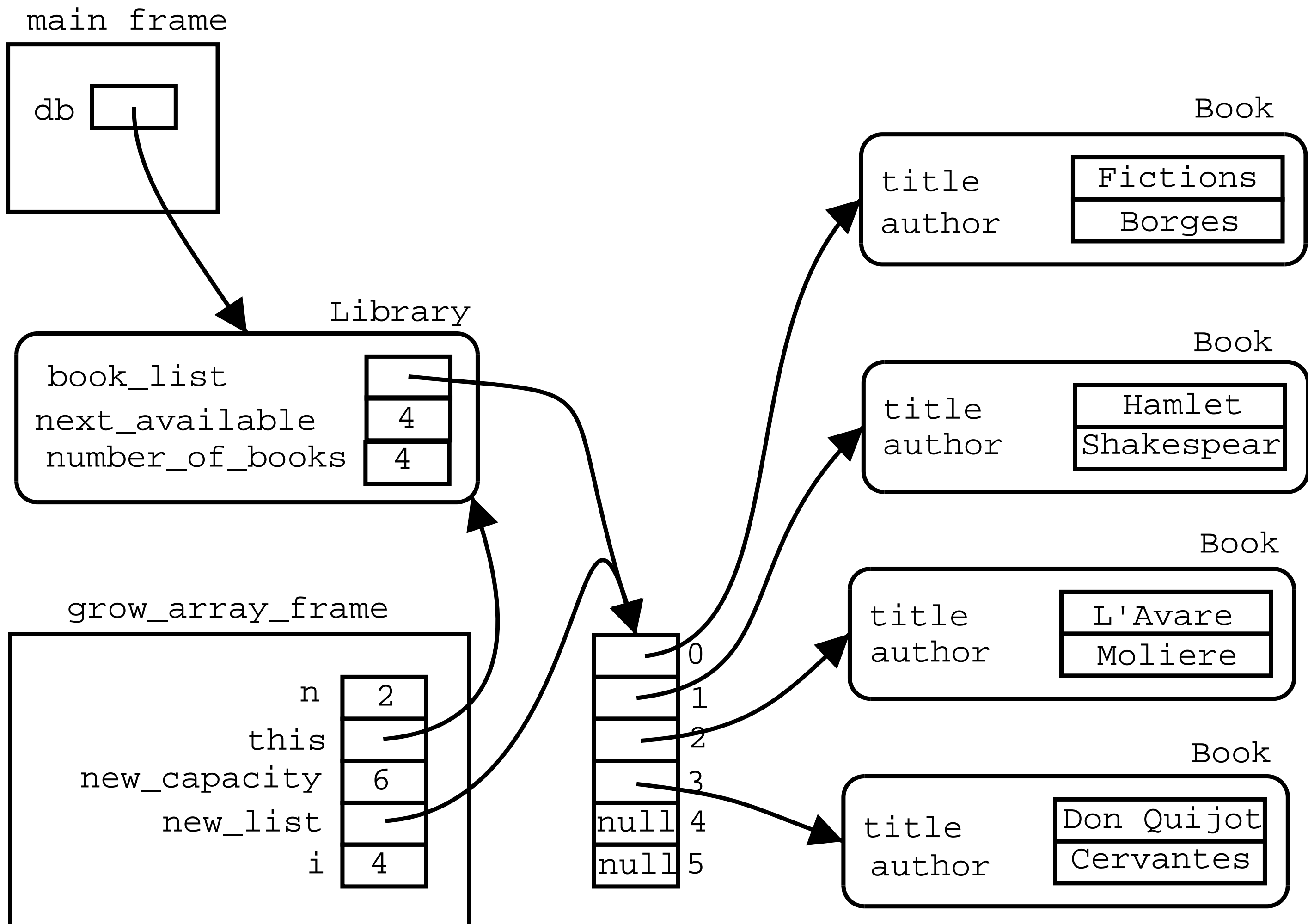
Growing arrays



Growing arrays

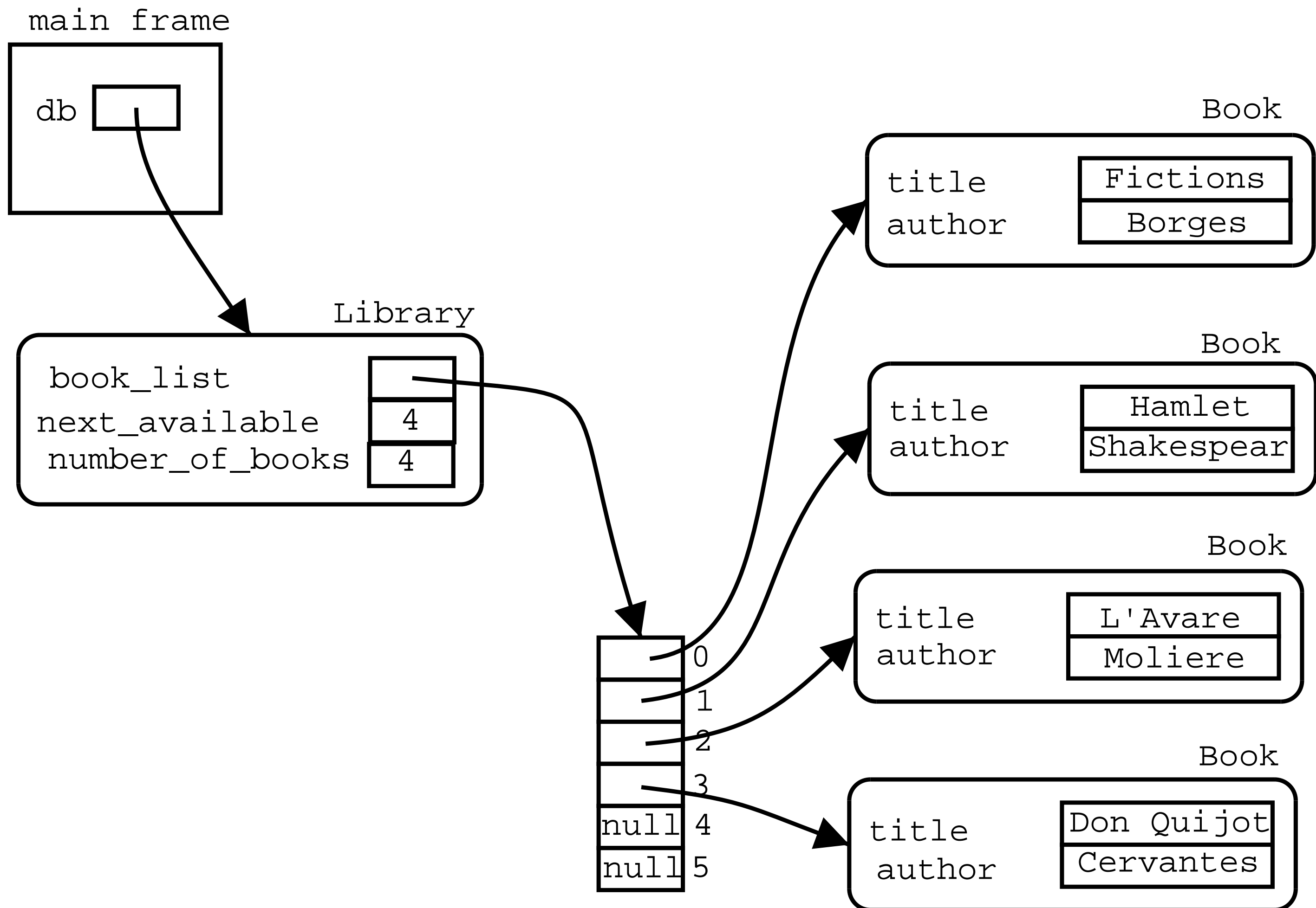


Growing arrays



McGill

Growing arrays



Array operations

- Adding elements
- Removing/deleting elements
- Finding elements
- Increasing the size of an array

Sorting

- Classical problem in Computer Science
- Problem: Given an array of objects, sort the array by some *key*.
- For example: Sort an array of students by name, or sort an array of products by price.
- Solution for small arrays using only conditionals is not *scalable*.

Sorting

- Analysis:
  - Objects:
    * An array of objects
  - Relationships:
    * Each object *has a* key (and maybe other attributes.)
    * For example, if the objects are of class Student, the key can be the name, to sort by name, or the id, to sort by id.
    * Each pair of keys can be compared: there is a (total) order relation between the keys.
  - Input: the array
  - Output: the array, or a copy, where the objects are placed in order (ascending) with respect the the key of interest.

- Small variation of the problem: sort an array of numbers: the order relation between keys is simply $<=$.

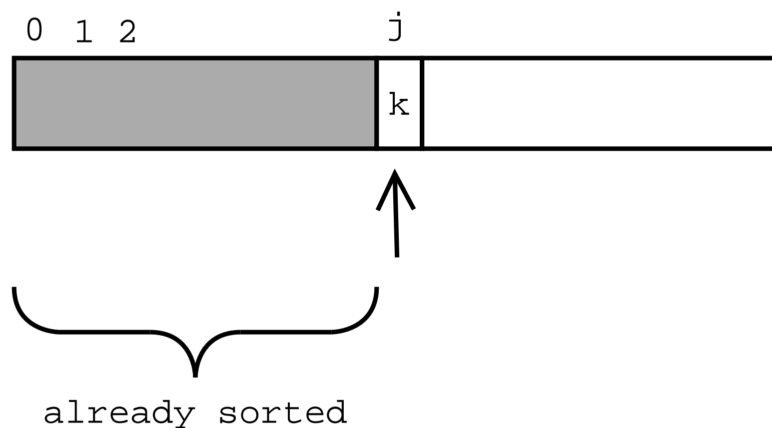
Sorting algorithms

- Insertion sort
- Selection sort
- Bubble sort
- Heap sort
- Merge sort
- Quick sort
- Bucket sort
- Counting sort
- Radix sort
- Sorting networks

Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

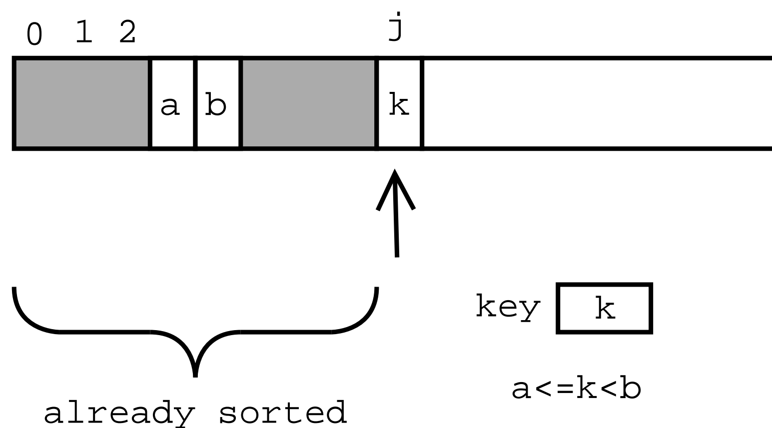
- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

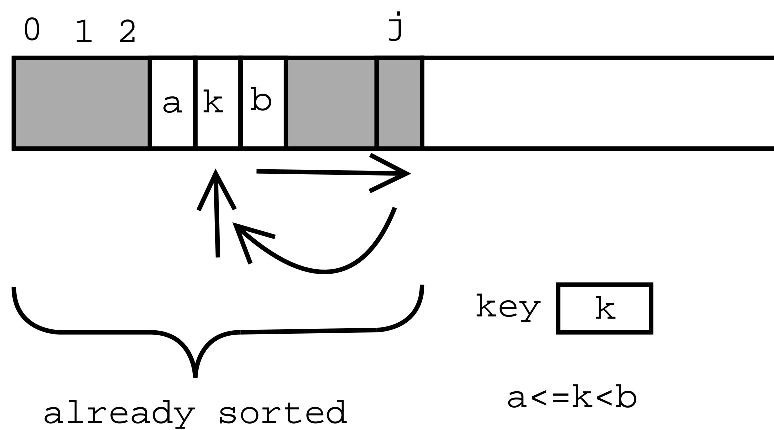
- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

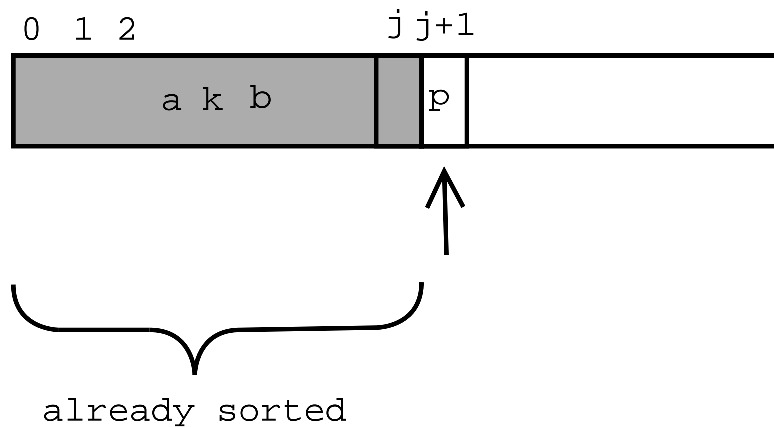
- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



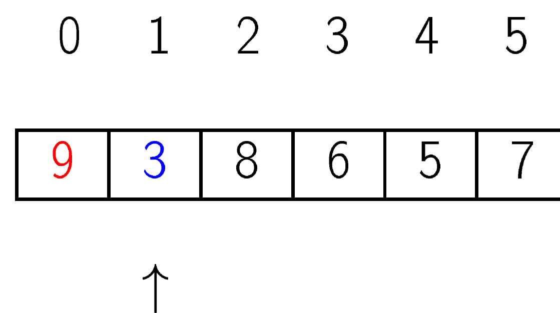
Insertion sort

- Example:

0	1	2	3	4	5
9	3	8	6	5	7

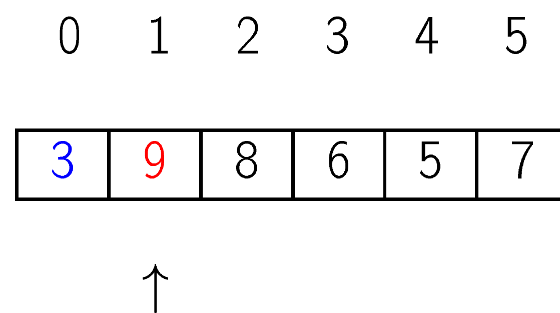
Insertion sort

- Example:



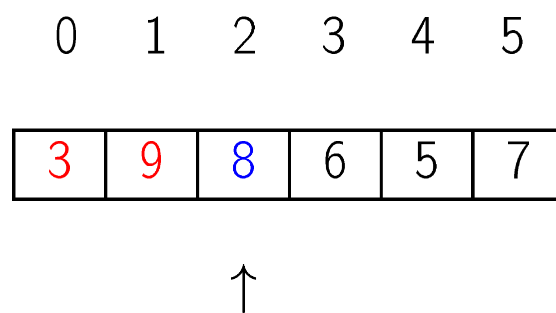
Insertion sort

- Example:



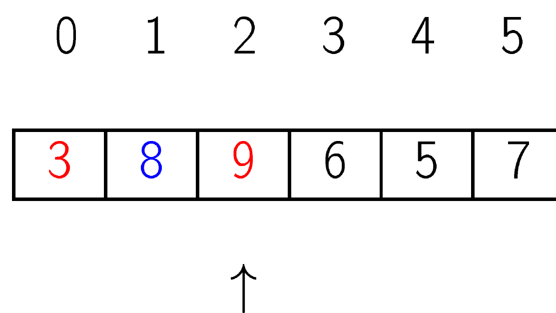
Insertion sort

- Example:



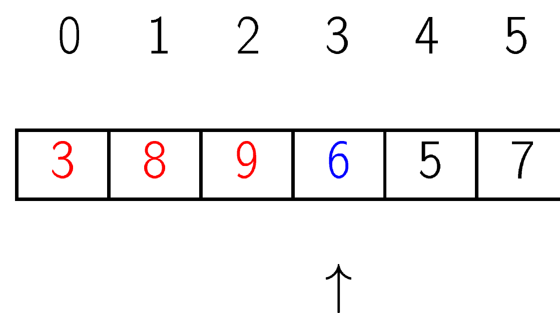
Insertion sort

- Example:



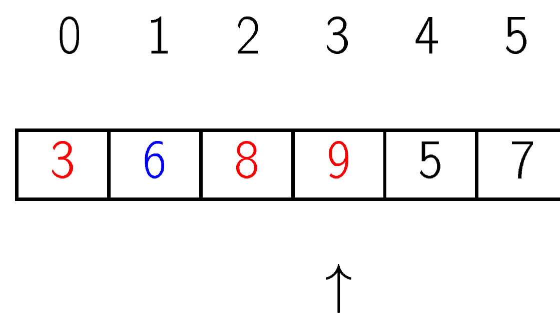
Insertion sort

- Example:



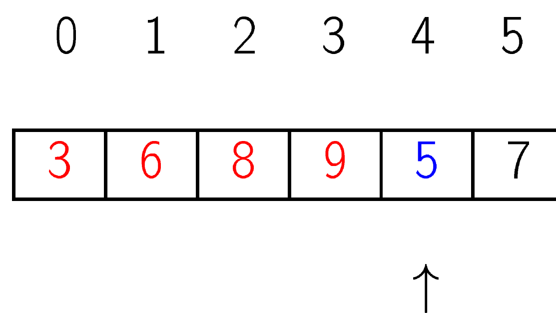
Insertion sort

- Example:



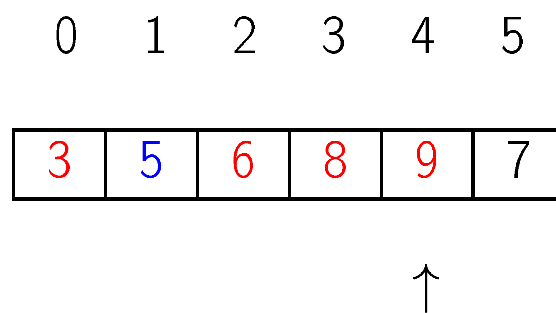
Insertion sort

- Example:



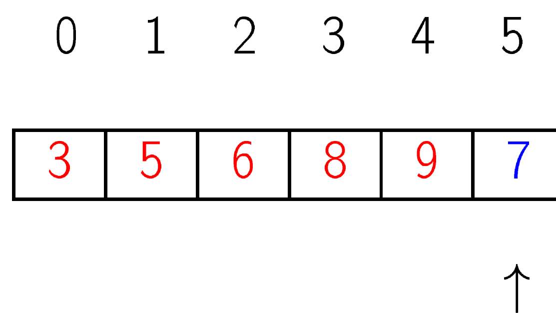
Insertion sort

- Example:



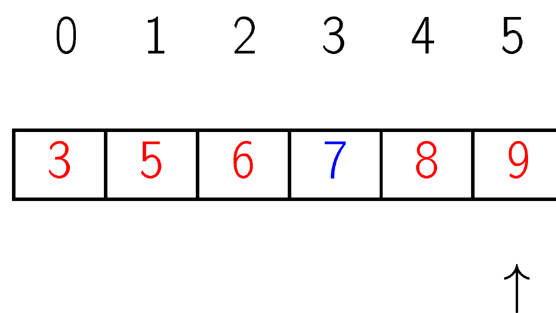
Insertion sort

- Example:



Insertion sort

- Example:



Insertion sort

- Example:

0	1	2	3	4	5
3	5	6	7	8	9

Insertion sort

- Algorithm:
  - Input: an array of numbers `a`

1. If `a[1]<a[0]` swap them.
2. Insert `a[2]` into `a[0..1]`
3. Insert `a[3]` into `a[0..2]`
4. Insert `a[4]` into `a[0..3]`
5. ...
6. Insert `a[length of a-1]` into `a[0..length of a-2]`

Insertion sort

- Algorithm refined:

1. For each j from 1 to the length of a-1
   (a) Insert a[j] into the sorted subarray a[0..j-1]

Insertion sort

- Algorithm refined: (Full algorithm)

1. For each `j` from 1 to the length of `a-1`
   (a) Set `key` to `a[j]`
   (b) Set `i` to `j - 1`
   (c) While `i >= 0` and `a[i] > key` do
      i. Set `a[i+1]` to `a[i]`
      ii. Decrement `i` by 1
   (d) Set `a[i+1]` to `key`

Insertion sort

- Implementation

```
void insertion_sort(int[] a)
{
  int i, j, key;
  j = 1;
  while (j < a.length)
  {
    key = a[j];
    i = j - 1;
    while (i >= 0 && a[i] > key)
    {
      a[i+1] = a[i];
      i--;
    }
    a[i+1] = key;
    j++;
  }
}
```

Sorting arrays of Objects

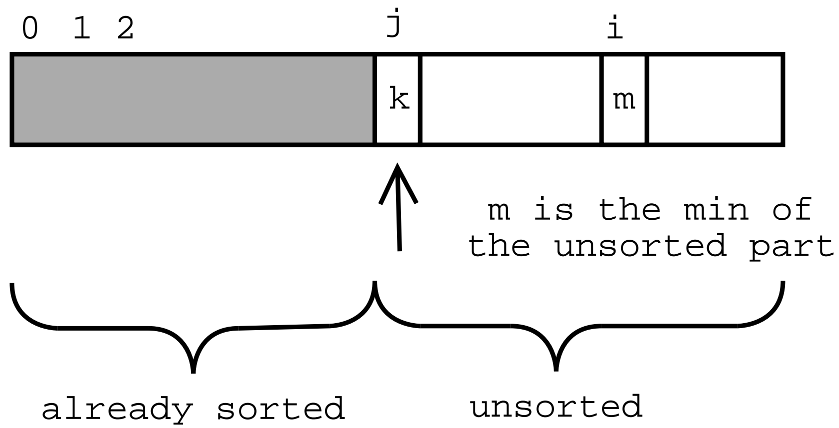
- Comparing strings: Lexicographical order
- The compareTo method from the String class
- s1.compareTo(s2) returns a negative integer if s1 is lexicographically before s2, 0 if they are equal, and a positive integer if s1 is lexicographically after s2.

```
String s1 = “aacb”, s2 = “aafa”;
int n = s1.compareTo(s2); // n = -3;
String s3 = “aacbgg”;
int m = s3.compareTo(s2); // n = -3
int k = s3.compareTo(s1); // n = 2
```

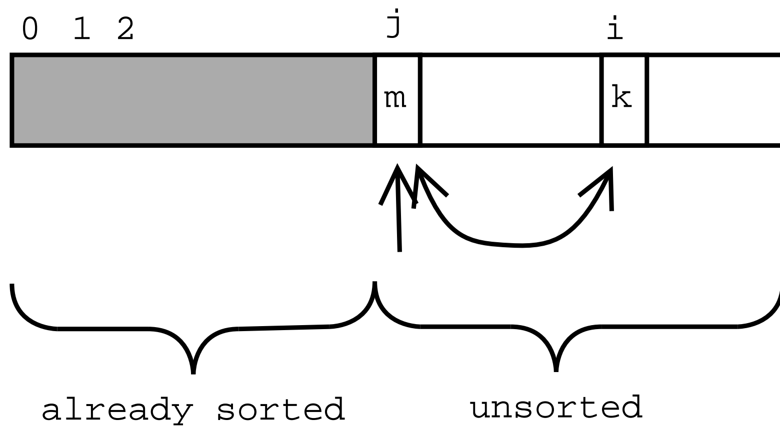
Sorting arrays of Objects

```
void insertion_sort() // In class Library
{
  int i, j;
  String key;
  Book focus;
  j = 1;
  while (j < book_list.length)
  {
    focus = book_list[j];
    key = focus.title();
    i = j - 1;
    while (i >= 0 &&
      key.compareTo(book_list[i].title()) < 0 )
    {
      book_list[i+1] = book_list[i];
      i--;
    }
    book_list[i+1] = focus;
    j++;
  }
}
```

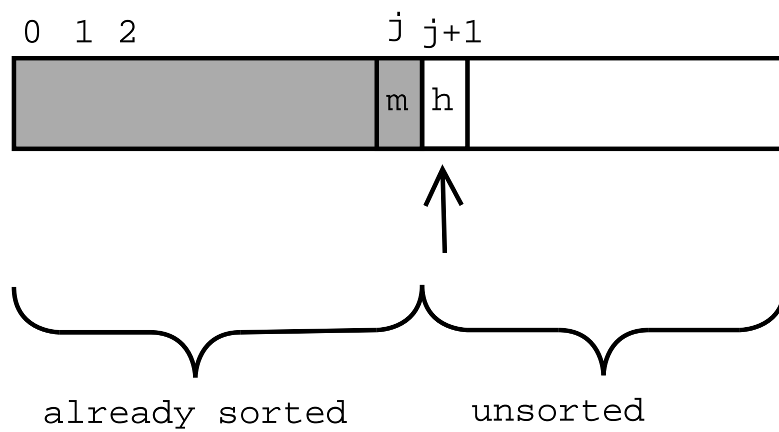
Selection sort



Selection sort



Selection sort



Selection sort

- Example:

0	1	2	3	4	5
9	3	8	6	5	7

Selection sort

- Example:

0	1	2	3	4	5
9	3	8	6	5	7
↑					

McGill

Selection sort

- Example:

0	1	2	3	4	5
9	3	8	6	5	7
$\uparrow$	$\Uparrow$				

Selection sort

- Example:

0	1	2	3	4	5
3	9	8	6	5	7
$\uparrow$	$\Uparrow$				

Selection sort

- Example:

0	1	2	3	4	5
3	9	8	6	5	7
	↑				

Selection sort

- Example:

0	1	2	3	4	5
3	9	8	6	5	7
	$\uparrow$			$\Uparrow$	

Selection sort

- Example:

0	1	2	3	4	5
3	5	8	6	9	7
	$\uparrow$			$\Uparrow$	

Selection sort

- Example:

0	1	2	3	4	5
3	5	8	6	9	7
		↑			

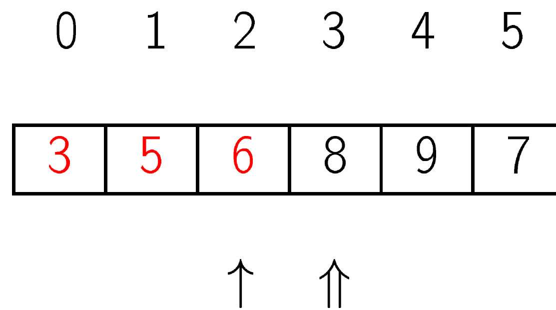
Selection sort

- Example:

0	1	2	3	4	5
3	5	8	6	9	7
		$\uparrow$	$\Uparrow$		

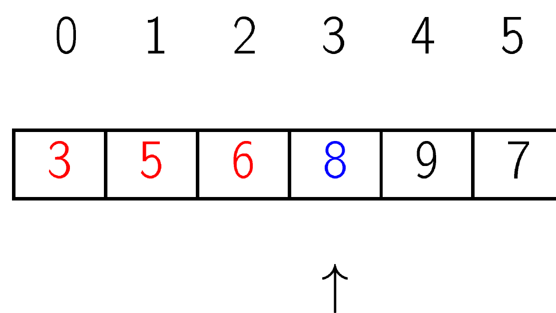
Selection sort

- Example:



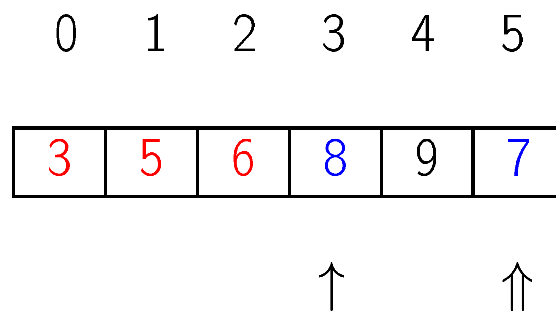
Selection sort

- Example:



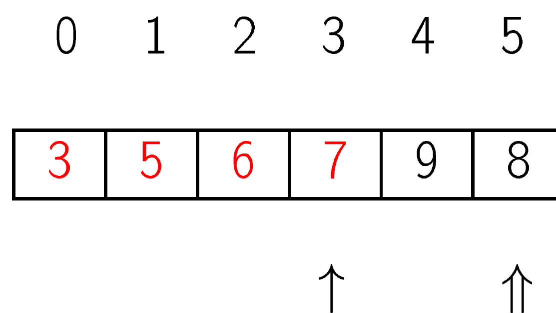
Selection sort

- Example:



Selection sort

- Example:



Selection sort

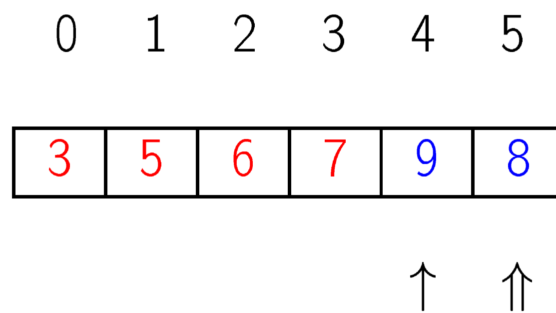
- Example:

0	1	2	3	4	5
3	5	6	7	9	8
				↑	

McGill

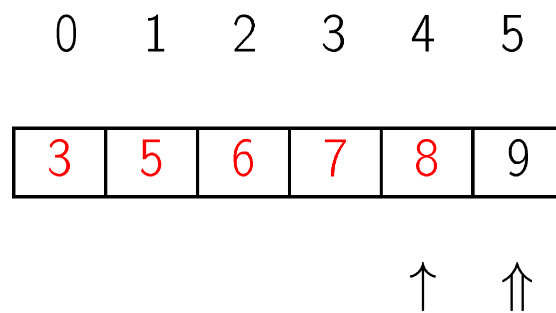
Selection sort

- Example:



Selection sort

- Example:



Selection sort

- Example:

0	1	2	3	4	5
3	5	6	7	8	9

Selection sort

- Idea:

1. Look for the minimum `m0` in `a[1..length a-1]`.
2. Swap the minimum and `a[0]`.
3. Look for the minimum `m1` in `a[2..length a-1]`
4. Swap `m1` with `a[1]`
5. Look for the minimum `m2` in `a[3..length a-1]`
6. Swap `m2` with `a[2]`
7. Look for the minimum `m3` in `a[4..length a-1]`
8. Swap `m3` with `a[3]`
9. ...

Selection sort

- Algorithm

1. For each j from 0 to length a - 2 do
   (a) Let min_index to be the index of the minimum in a[j+1..length a-1]
   (b) Swap a[min_index] and a[j]

Selection sort

- Algorithm refined

1. For each `j` from `0` to length `a - 2` do
   (a) Let `minimum` be `a[j]`
   (b) Set `min_index` to `j`
   (c) For each `i` from `j+1` to the length `a - 1` do
      i. If `a[i] < minimum` then
         A. Set `minimum` to `a[i]`
         B. Set `min_index` to `i`
   (d) Swap `a[min_index]` and `a[j]`

Selection sort

```
void selection_sort(int[] a)
{
  int minimum, min_index, temp;
  int j = 0, i;
  while (j <= a.length - 2)
  {
    minimum = a[j];
    min_index = j;
    i = j + 1;
    while (i <= a.length - 1) {
      if (a[i] < minimum) {
        minimum = a[i];
        min_index = i;
      }
      i++;
    }
    temp = a[j];
    a[j] = a[min_index];
    a[min_index] = temp;
    j++;
  }
}
```

Selection sort

```
void selection_sort() {
  int min_index, j = 0, i;
  String minimum;
  Book temp;
  while (j <= a.length - 2) {
    minimum = book_list[j].title();
    min_index = j;
    i = j + 1;
    while (i <= book_list.length - 1) {
      String current_key = book_list[i].title();
      if (current_key.compareTo(minimum) < 0) {
        minimum = current_key;
        min_index = i;
      }
      i++;
    }
    temp = book_list[j];
    book_list[j] = book_list[min_index];
    book_list[min_index] = temp;
    j++;
  }
}
```

Linear search

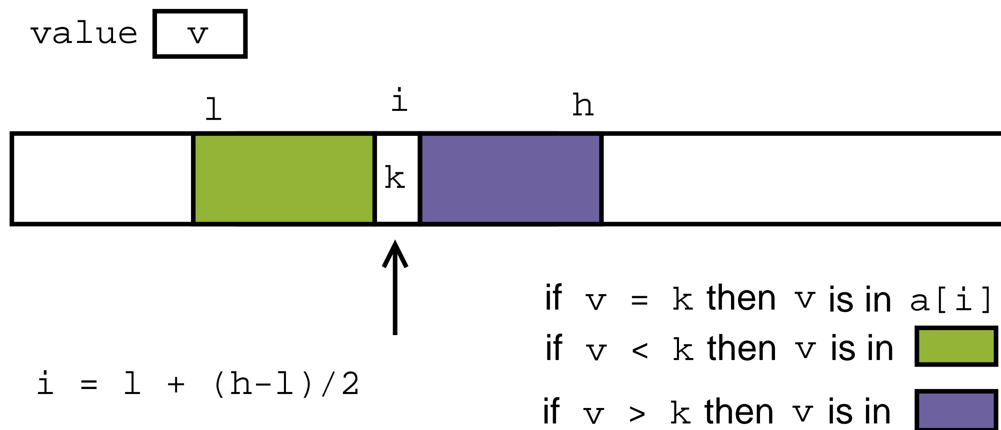
```
int linear_search(int[] a, int value)
{
  int index = 0;
  while (index < a.length)
  {
    if (value == a[index])
    {
      return index;
    }
    index++;
  }
  return -1;  // Not found
}
```

- This works for unsorted arrays

Binary search

- But if we know that the array is sorted, we can improve the speed of searching by ignoring parts which do not need to look at.

- If we are looking for a value `v` in an array `a`, and we have already narrowed down the search space to `a[l..h]`, then



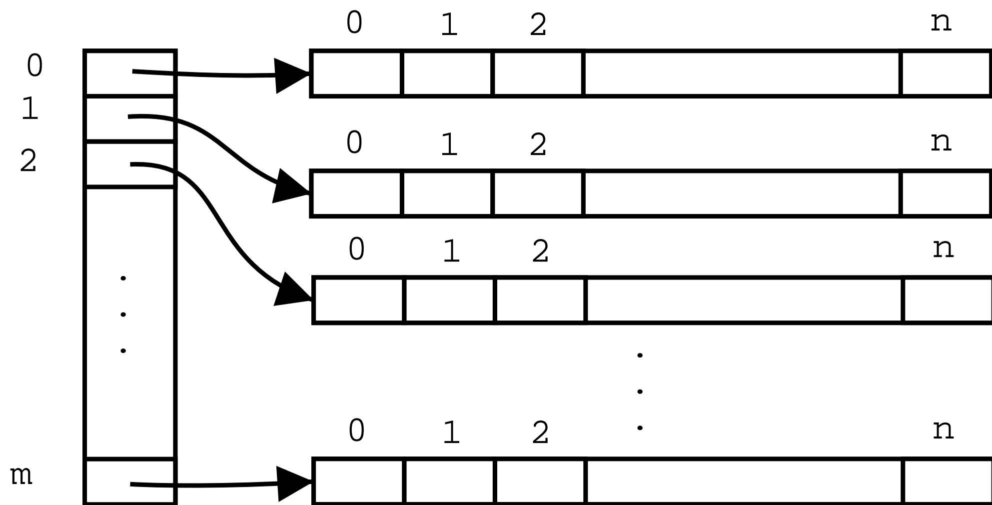
Binary search

```
int binary_search(int[] a, int value)
{
  int lower = 0, higher = a.length - 1, index;
  while (lower <= higher)
  {
    index = lower + (higher - lower) / 2;
    if (value == a[index]) {
      return index;
    }
    else if (value < a[index]) {
      higher = index - 1;
    }
    else {  // value > a[index]
      lower = index + 1;
    }
  }
  return -1;  // Not found
}
```

Binary search

```
int binary_search(String title)
{
  int lower = 0, higher = book_list.length - 1;
  int index;
  String current_title;
  int comparison;
  while (lower <= higher) {
    index = lower + (higher - lower) / 2;
    current_title = book_list[index].title();
    comparison = title.compareTo(current_title);
    if (comparison == 0) {
      return index;
    }
    else if (comparison < 0) {
      higher = index - 1;
    }
    else {  // comparison > 0
      lower = index + 1;
    }
  }
  return -1;  // Not found
}
```

Multidimensional arrays



	0	1	2		n
0				...	
1				...	
2				...	
	⋮	⋮	⋮	⋮	⋮
m				...	

Multidimensional arrays

- A two-dimensional array is an array of arrays.

```
int[][] table = new int[5][10];

for (int row=0; row < table.length; row++)
  for (int col=0; col < table[row].length; col++)
    table[row][col] = row * 10 + col;
```

- A multidimensional array is an n-dimensional array, i.e. an array of arrays of arrays of ...

- Processing nested arrays is commonly done with nested loops.

Multidimensional arrays

- A two-dimensional array is an array of arrays.

```
int[][] table = new int[5][10];

for (int row=0; row < table.length; row++)
  for (int col=0; col < table[row].length; col++)
    table[row][col] = row * 10 + col;
for (int row=0; row < table.length; row++) {
  for (int col=0; col < table[row].length; col++)
    System.out.print(table[row][col]+"\t");
  System.out.println();
}
```

- A multidimensional array is an n-dimensional array, i.e. an array of arrays of arrays of ...

- Processing nested arrays is commonly done with nested loops.

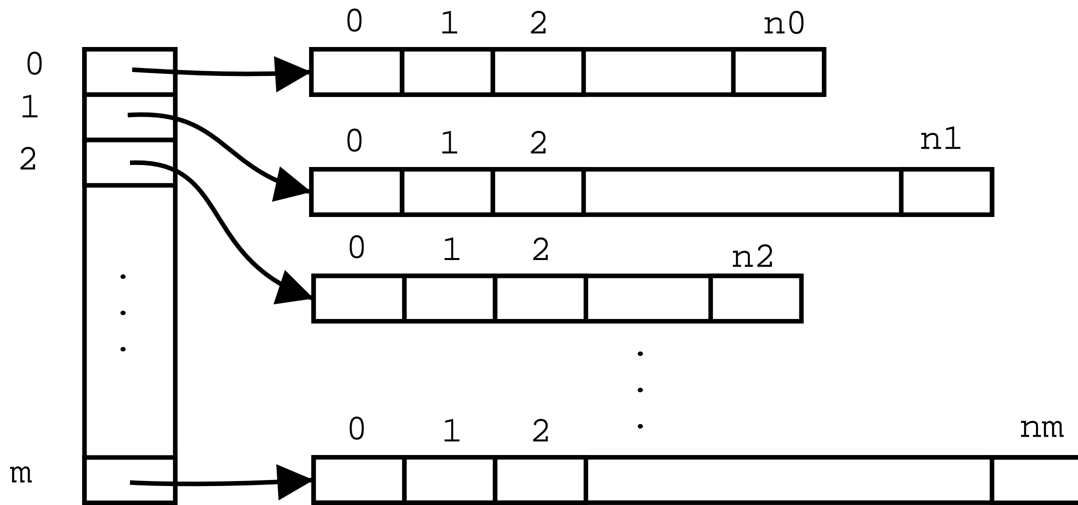
Multidimensional arrays

- A two-dimensional array can be an array of objects

```
class A { int x; }

// and in the client
A[][] table = new A[5][10];
for (int row=0; row < table.length; row++)
  for (int col=0; col < table[row].length; col++) {
    table[row][col] = new A();
    table[row][col].x = row * 10 + col;
  }
```

Multidimensional arrays



Multidimensional arrays

- Each row can have different length

```
class A { int x; }

// and in the client
A[][] table = new A[5][];
for (int row=0; row < table.length; row++)
  for (int col=0; col < table[row].length; col++)
    table[row][col] = new A();
    table[row][col].x = row * 10 + col;
  }
```

Efficiency

- Linear search: $O(n)$

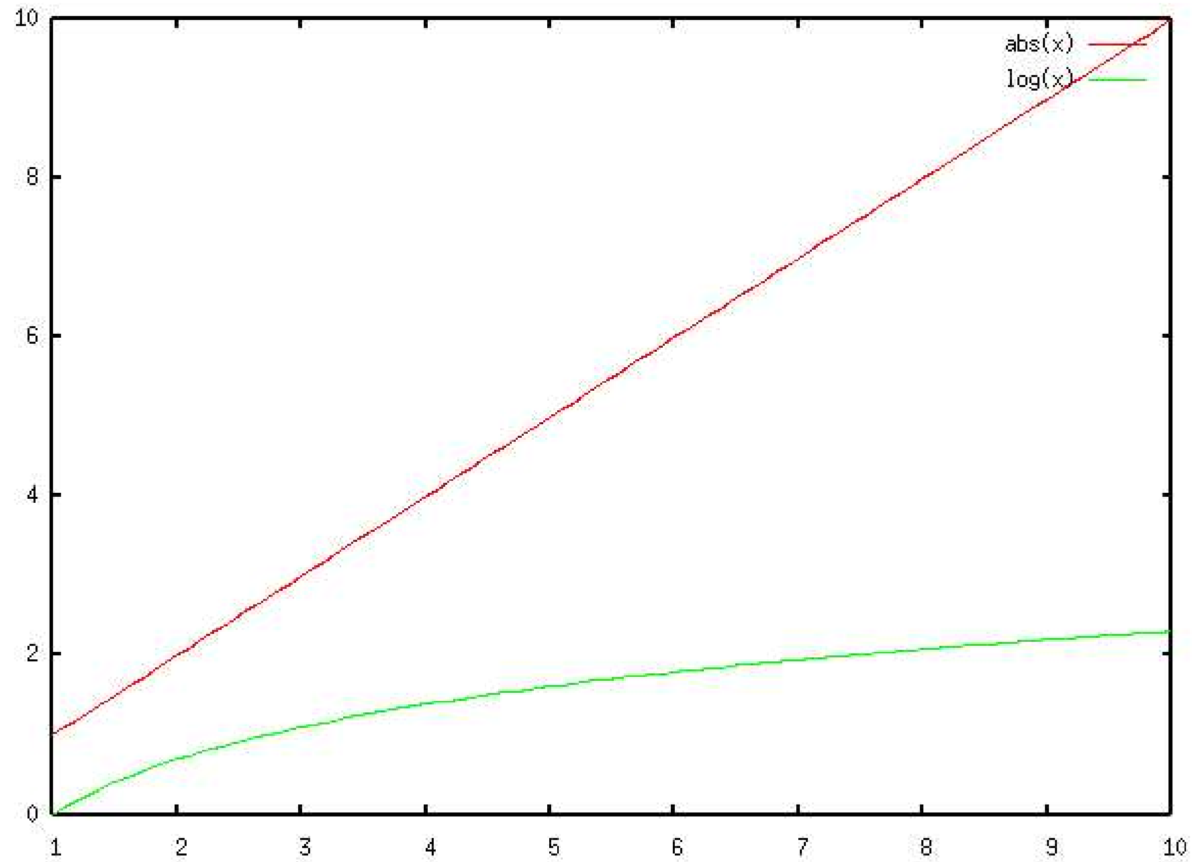
n	# of comparisons
1	1
2	2
3	3
4	4
⋮	⋮
1000	1000
⋮	⋮
10000000	10000000
⋮	⋮
k	k

Efficiency

- Binary search: $O(log_2 n)$

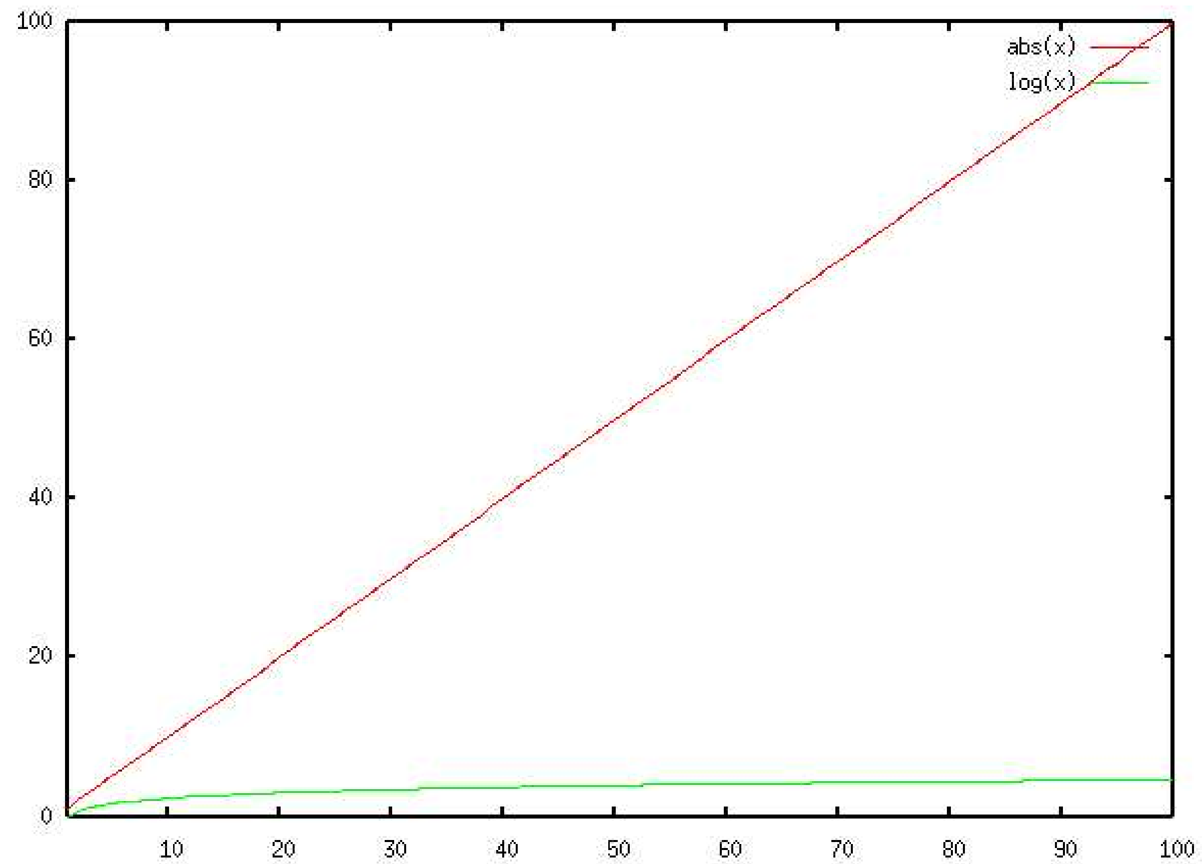
n	# of comparisons
1	0
2	2
3	3
⋮	⋮
10	4
100	7
1000	10
10000	14
100000	17
1000000	20
10000000	24
100000000	27
10^9	30
10^{10}	33

Efficiency

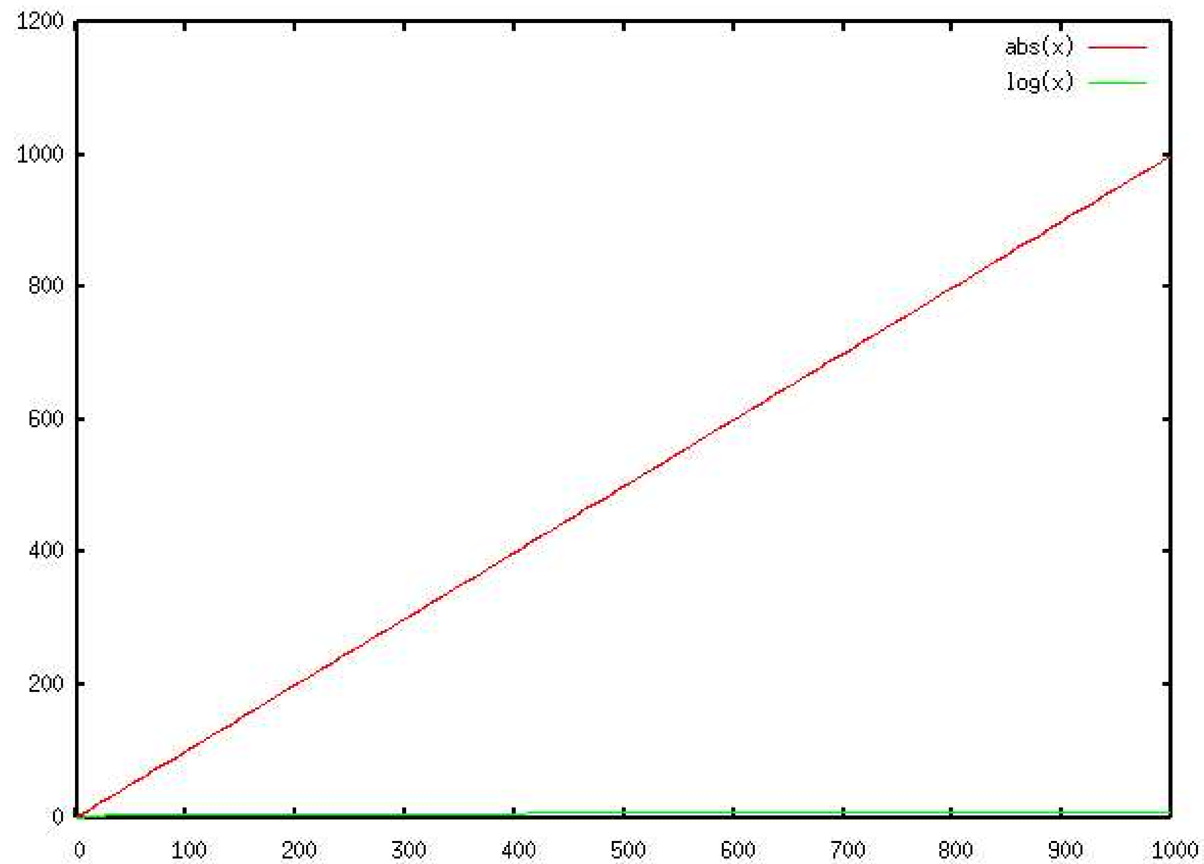


McGill

Efficiency



Efficiency



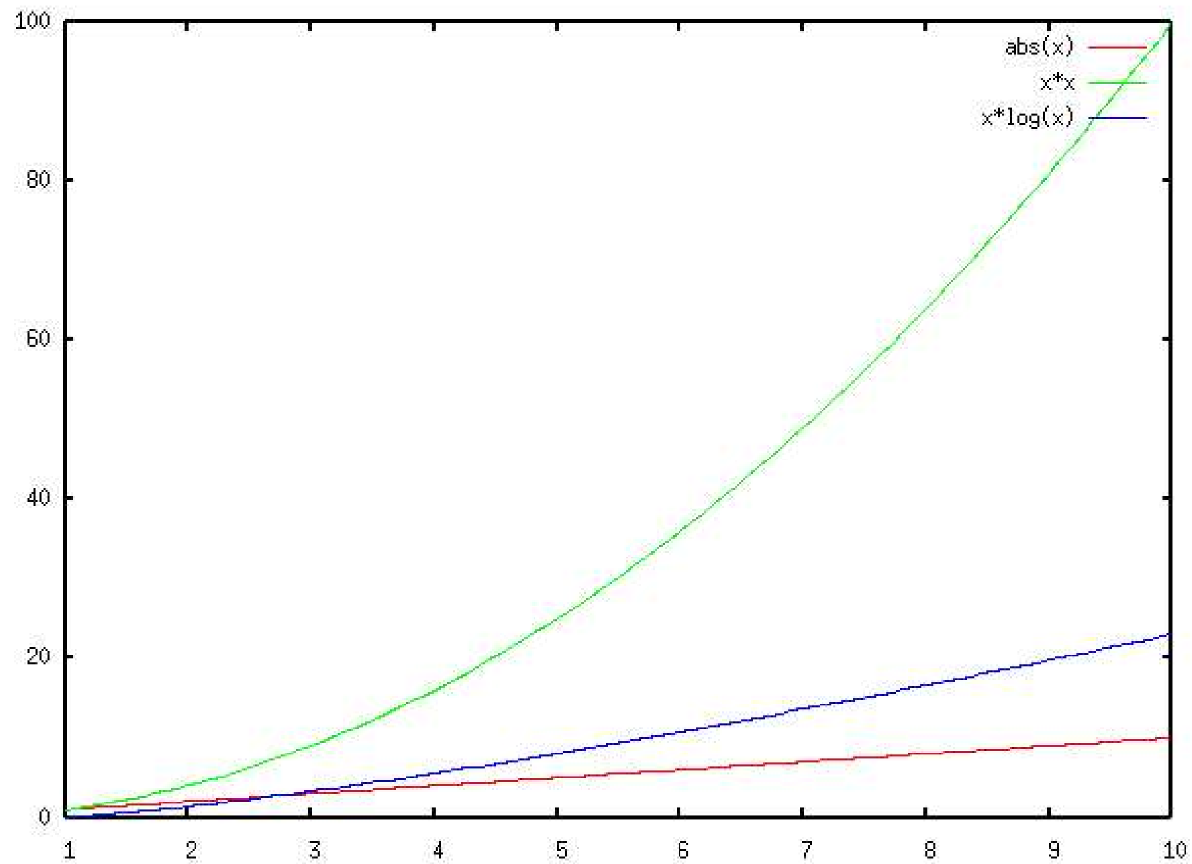
Sorting algorithms

- Insertion sort
- Selection sort
- Bubble sort
- Heap sort
- Merge sort
- Quick sort
- Bucket sort
- Counting sort
- Radix sort
- Sorting networks

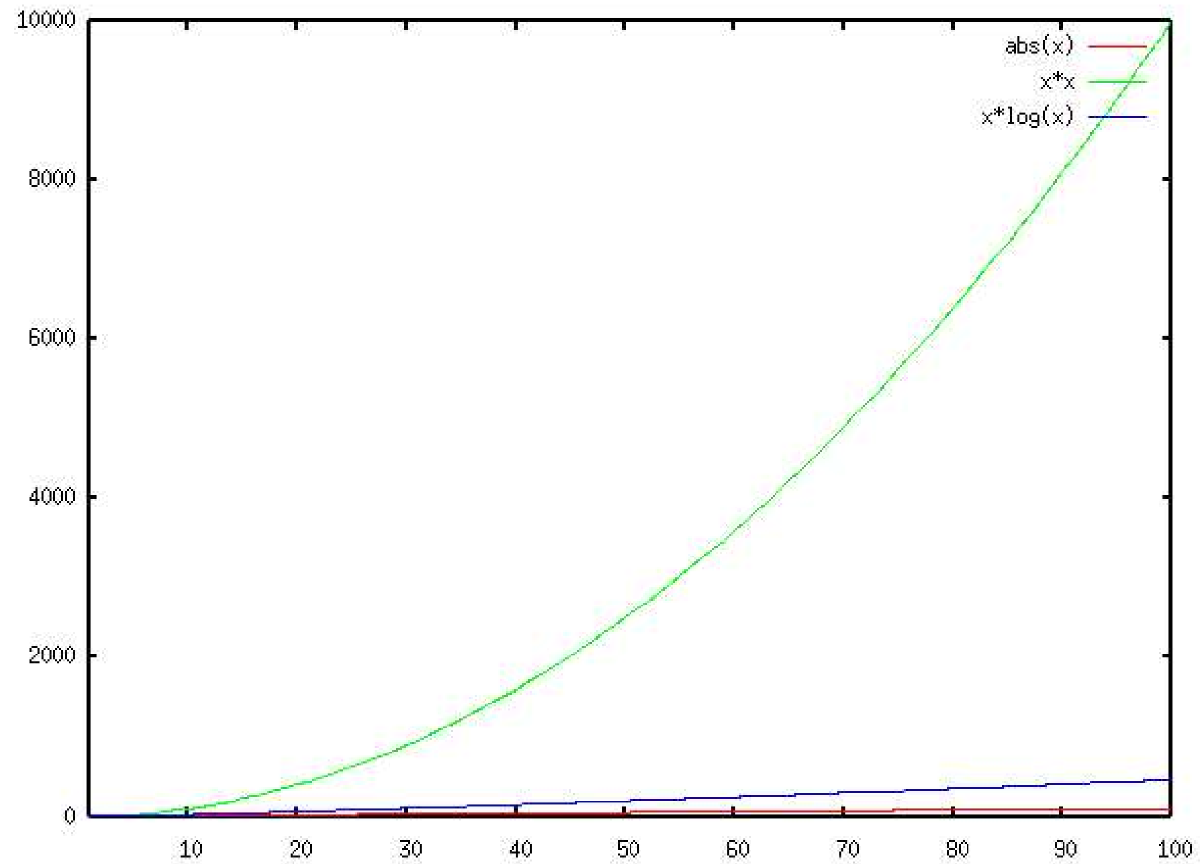
Sorting algorithms

Algorithm	Complexity	$n = 1000$	$n = 10^6$
Insertion sort	$O(n^2)$	10^6	10^{12}
Selection sort	$O(n^2)$	10^6	10^{12}
Bubble sort	$O(n^2)$	10^6	10^{12}
Heap sort	$O(n \log_2 n)$	≈ 10000	$\approx 20 \times 10^7$
Merge sort	$O(n \log_2 n)$	≈ 10000	$\approx 20 \times 10^7$
Quick sort	$O(n^2)$ in the worst case, but $O(n \log_2 n)$ on average		
Bucket sort	$O(n)$ but with restrictions	1000	10^6
Counting sort	$O(n)$ but with restrictions	1000	10^6
Radix sort	$O(n)$ but with restrictions	1000	10^6
Sorting networks	$O(n)$ but with restrictions	1000	10^6

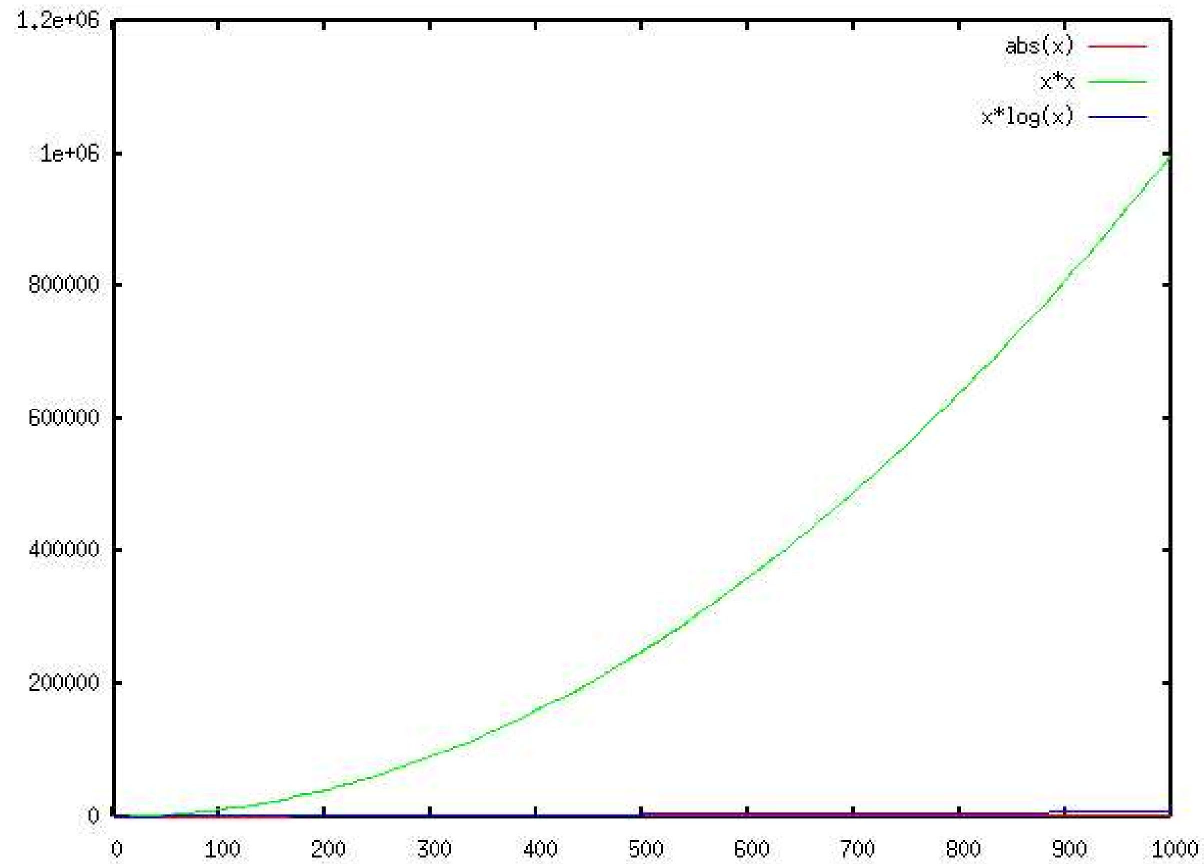
Efficiency



Efficiency



Efficiency



The end