SORTING AND SEARCHING

Lecture notes of the course "*Programming Techniques*"

Lê Hồng Phương[1]

[1]Department of Mathematics, Mechanics and Informatics
VNU University of Science, Hanoi
<*phuonglh@gmail.com*>

09/2012

Content

1. Sorting
   - Insertion sort
   - Other sorting algorithms

2. Searching
   - Linear search
   - Binary search

3. Exercises

Content

1. Sorting
   - Insertion sort
   - Other sorting algorithms
2. Searching
   - Linear search
   - Binary search
3. Exercises

Content

1. Sorting
   - Insertion sort
   - Other sorting algorithms
2. Searching
   - Linear search
   - Binary search
3. Exercises

Content

1. Sorting
   - Insertion sort
   - Other sorting algorithms

2. Searching
   - Linear search
   - Binary search

3. Exercises

Insertion sort

- Insertion sort is a simple sorting algorithm.
- Like bubble sort, it is mostly efficient for small data sets and is often used as part of more sophisticated algorithms.
- **How does it work?**
  - Take elements from the array one by one
  - Insert them to their correct position
- Insertion is expensive because it requires shifting all following elements over by one.

Insertion sort

- Insertion sort is a simple sorting algorithm.
- Like bubble sort, it is mostly efficient for small data sets and is often used as part of more sophisticated algorithms.
- **How does it work?**
  - Take elements from the array one by one
  - Insert them to their correct position
- Insertion is expensive because it requires shifting all following elements over by one.

Insertion sort

A simple insertion sort: $O(n^2)$

- Iterate through array until an out-of-order element found
- Insert out-of-order element into correct location
- Repeat until end of array reached

Split into two functions for ease-of-use

- `shiftElement()`
- `insertionSort()`

Insertion sort

A simple insertion sort: $O(n^2)$

- Iterate through array until an out-of-order element found
- Insert out-of-order element into correct location
- Repeat until end of array reached

Split into two functions for ease-of-use

- `shiftElement()`
- `insertionSort()`

Shift elements

Shift elements on the left of a position i to find the correct position for $a[i]$:

```
void shiftElement(int a[], int i) {
  int iValue = a[i];
  while ((i > 0) && (a[i-1] > iValue)) {
    a[i] = a[i-1];
    i--;
  }
  a[i] = iValue;
}
```

Shift elements

Example: $i = 3, a = \{17, 25, 31, 18, 2\}$, `iValue = 18`.

i	0	1	2	3	4
$a[i]$	17	25	31	18	2
				$\uparrow$	

$\Rightarrow$

i	0	1	2	3	4
$a[i]$	17	25	**31**	31	2
			$\uparrow$		

i	0	1	2	3	4
$a[i]$	17	**25**	25	31	2
		$\uparrow$			

$\Rightarrow$

i	0	1	2	3	4
$a[i]$	17	**25**	25	31	2
		$\uparrow$			

i	0	1	2	3	4
$a[i]$	17	18	25	31	2
		$\uparrow$			

Run the function with $i = 4$. What does the array look like?

Shift elements

Example: $i = 3, a = \{17, 25, 31, 18, 2\}$, `iValue = 18`.

i	0	1	2	3	4
$a[i]$	17	25	31	18	2
				$\uparrow$	

$\Rightarrow$

i	0	1	2	3	4
$a[i]$	17	25	**31**	31	2
			$\uparrow$		

i	0	1	2	3	4
$a[i]$	17	**25**	25	31	2
		$\uparrow$			

$\Rightarrow$

i	0	1	2	3	4
$a[i]$	17	**25**	25	31	2
		$\uparrow$			

i	0	1	2	3	4
$a[i]$	17	18	25	31	2
		$\uparrow$			

Run the function with $i = 4$. What does the array look like?

Insertion sort

Simply shift each element of the array if it is not in the correct position.

```
void insertionSort(int a[], int n) {
  int i;
  for (i = 1; i < n; i++) {
    if (a[i] < a[i-1]) {
      shiftElement(a, i);
    }
  }
}
```

Insertion sort

- Explain the algorithm on the following arrays:

$$a = \{25, 17, 31, 18, 2\}$$
$$a = \{3, 2, 1, 5, 1, 6\}$$

- Rewrite the algorithm using pointer arithmetic instead of indexing.

Content

1. Sorting
   - Insertion sort
   - Other sorting algorithms
2. Searching
   - Linear search
   - Binary search
3. Exercises

Other sorting algorithms

- We have seen 3 simple sorting algorithms. They are **not** efficient when the data set is large.
- There exists faster and more efficient sorting algorithms: quicksort, heapsort, shell sort, merge sort. . .
- We shall present these algorithms later when related programming techniques are discussed:
  - Quicksort uses recursive functions
  - Heapsort uses heap data structure

Content

Linear search – Indexing version

Search for an element in an array:

```
int* linearSearch(int *a, int n, int value) {
  int i;
  for (i = 0; i < n; i++) {
    if (a[i] == value) {
      return a + i;
    }
  }
  return NULL;
}
```

- `n` is the number of elements of the array
- `value` is the value to search for (key)
- The function returns a pointer to the element found or NULL

Linear search – Pointer arithmetic version

```
int* linearSearch(int *a, int n, int value) {
  int *pa;
  for (pa = a; pa < a + n; pa++) {
    if (*pa == value) {
      return pa;
    }
  }
  return NULL;
}
```

Content

Binary search

- If the array is already sorted, a better (faster) search algorithm is binary search.
- A binary search (or half-interval search) finds the position of a specified within a sorted array.
- How does it work?

Binary search

At each stage, the algorithm compares the input key value with the value of the middle element of the array.

- If the key matches, its position is returned.
- If the key is less than the middle element's key, the algorithm repeats its action on the sub-array to the left of the middle element.
- If the key is greater than the middle element's key, the algorithm repeats its action on the sub-array to the right of the middle element.
- If the remaining array to be searched is reduced to zero, then the key cannot be found.

Binary search

At each stage, the algorithm compares the input key value with the value of the middle element of the array.

- If the key matches, its position is returned.
- If the key is less than the middle element's key, the algorithm repeats its action on the sub-array to the left of the middle element.
- If the key is greater than the middle element's key, the algorithm repeats its action on the sub-array to the right of the middle element.
- If the remaining array to be searched is reduced to zero, then the key cannot be found.

Binary search

At each stage, the algorithm compares the input key value with the value of the middle element of the array.

- If the key matches, its position is returned.
- If the key is less than the middle element's key, the algorithm repeats its action on the sub-array to the left of the middle element.
- If the key is greater than the middle element's key, the algorithm repeats its action on the sub-array to the right of the middle element.
- If the remaining array to be searched is reduced to zero, then the key cannot be found.

Binary search

At each stage, the algorithm compares the input key value with the value of the middle element of the array.

- If the key matches, its position is returned.
- If the key is less than the middle element's key, the algorithm repeats its action on the sub-array to the left of the middle element.
- If the key is greater than the middle element's key, the algorithm repeats its action on the sub-array to the right of the middle element.
- If the remaining array to be searched is reduced to zero, then the key cannot be found.

Binary search

```
int* binarySearch(int *a, int n, int value) {
  int i, j, m;
  i = 0; j = n - 1;
  while (i <= j) {
    m = (i+j)/2;
    if (a[m] == value) {
      return a + m;
    } else {
      if (a[m] < value) {
        i = m + 1;
      } else {
        j = m-1;
      }
    }
  }
  return NULL;
}
```

Binary search – `bsearch()` in `stdlib.h`

The prototype for the function `bsearch()`:

```
void* bsearch(const void *value, const void *array,
        size_t arrayLength, size_t size,
    int(*comparator)(const void *, const void *));
```

- `value` is the value to search for (key).
- `array` is the array which is *sorted in increasing order*.
- `size_t` is an unsigned integral type.
- `arrayLength` is the number of elements in the array.
- `size` is the size of each element (in bytes) in the array.
- `comparator` is a *function* that compares two elements of the array.
- The function returns a pointer or null.

Comparator function

A general comparator function looks like this:

```
int comparator(const void *a, const void *b) {
  if (*(AType*)a > *(AType*)b) {
    return 1;
  }
  if (*(AType*)a == *(AType*)b) {
    return 0;
  }
  if (*(AType*)a < *(AType*)b) {
    return -1;
  }
}
```

Note: If you want to compare C strings you can directly use the function `strcmp()` as the comparator.

Comparator function

An integer comparator function:

```
int comparator(const void *a, const void *b) {
  if (*(int*)a > *(int*)b) {
    return 1;
  }
  if (*(int*)a == *(int*)b) {
    return 0;
  }
  if (*(int*)a < *(int*)b) {
    return -1;
  }
}
```

Comparator function

Why this integer comparator does not work?

```
int comparator(const void *a, const void *b) {
  return (*(int*)a - *(int*)b);
}
```

Using bsearch() on a sorted array

```
#include <stdlib.h>

int main() {
  int array[] = {1, 2, 5, 7, 12, 19};

  int value = 7;
  int *result = (int*)bsearch(&value, array,
      6, sizeof(int), comparator);
  if (result != NULL) {
    printf("%d is found in the array.\n", *result);
  }
  else {
    printf("%d is not found in the array.\n", value);
  }
  return 0;
}
```

Exercises

1. Implement the insertion sort algorithm.
2. Implement the linear search algorithm.
3. Implement the binary search algorithm.
4. Using the `bsearch()` function provided by `stdlib.h`.

For each of the algorithms above, you need to write two versions using two techniques, either indexing or pointer arithmetic, and

- Test your programs on data of different types (integers, double, strings). These data are read from a file.
- Use dynamic memory for allocating the arrays.