Searching & Sorting

Week 11

Gaddis: 8, 19.6, 19.8

CS 5301
Fall 2017

Jill Seaman

Definitions of Search and Sort

- **Search**: find a given item in a list, return the position of the item, or -1 if not found.
- **Sort**: rearrange the items in a list into some order (smallest to biggest, alphabetical order, etc.).
- “list” could be: array, linked list, string, etc.
- There are various methods (algorithms) for carrying out these common tasks.

Linear Search

- Compare first element to target value, if not found then compare second element to target value...
- Repeat until: target value is found (return its position) or we run out of items (return -1).

```java
int searchList (int list[], int size, int value) {
    for (int i=0; i<size; i++) {
        if (list[i] == value)
            return i;
    }
    return -1;
}
```

Other forms of Linear Search

- Recursive linear search over arrays
  - Gaddis ch 19, Prog Challenge #8: isMember
- Linear search over linked list
  - Gaddis ch 17, Prog Challenge #5: List search
- Recursive linear search over linked list
  - Another good exercise
Binary Search

- Works only for SORTED arrays
- Divide and conquer style algorithm
- Compare target value to middle element in list.
  - if equal, then return its index
  - if less than middle element, repeat the search in the first half of list
  - if greater than middle element, repeat the search in last half of list
- If current search list is narrowed down to 0 elements, return -1

**Binary Search Algorithm**

**example**

```
target is 11

{0} {1} {2} {3} {4} {5}  {6} {7} {8} {9} {10} {11} {12}
```

```
target < 50
```

```
list:  2  4  7 10 11 45  50  59 60 66 69 70  79
```

```
first mid last
```

```
target > 7
```

```
list:  2  4  7 10 11 45
```

```
first mid last
```

```
target == 11
```

```
list: 10 11 45
```

---

**Binary Search in C++**

Iterative version

```cpp
int binarySearch (int array[], int size, int target) {
    int first = 0,          //index to (current) first elem
        last = size - 1,    //index to (current) last elem
        middle,             //index of (current) middle elem
        position = -1;      //index of target value
    bool found = false;     //flag
    while (first <= last && !found) {
        middle = (first + last) /2;    //calculate midpoint
        if (array[middle] == target) {
            found = true;
            position = middle;
        } else if (target < array[middle]) {
            last = middle - 1; //search lower half
        } else {
            first = middle + 1; //search upper half
        }
    }
    return position;
}
```

---

Recursive version

```cpp
int binarySearchRec(int array[], int first, int last, int value) {
    int middle; // Mid point of search
    if (first > last)           //check for empty list
        return -1;
    middle = (first + last)/2;  //compute middle index
    if (array[middle]==value)
        return middle;
    if (value < array[middle])    //recursion
        return binarySearchRec(array, first,middle-1, value);
    else
        return binarySearchRec(array, middle+1,last, value);
}

int binarySearch(int array[], int size, int value) {
    return binarySearchRec(array, 0, size-1, value);
}
```
What is sorting?

• Sort: rearrange the items in a list into ascending or descending order
  - numerical order
  - alphabetical order
  - etc.

55 112 78 14 20 179 42 67 190 7 101 1 122 170 8
1 7 8 14 20 42 55 67 78 101 112 122 170 179 190

Selection Sort

• There is a pass for each position (0..size-1)
• On each pass, the smallest (minimum) element in the rest of the list is exchanged (swapped) with element at the current position.
• The first part of the list (the part that is already processed) is always sorted
• Each pass increases the size of the sorted portion.

Selection sort

Example

• 36 24 10 6 12  pass 1: minimum is 6, swap
• 6 24 10 36 12  pass 2: minimum is 10, swap
• 6 10 24 36 12  pass 3: minimum is 12, swap
• 6 10 12 36 24  pass 4: minimum is 24, swap
• 6 10 12 24 36  sorted

Note: first n elements are sorted after pass n

Selection sort: code

```c
// Returns the index of the smallest element, starting at start
int findIndexOfMin (int array[], int size, int start) {
    int minIndex = start;
    for (int i = start+1; i < size; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}

// Sorts an array, using findIndexOfMin
void selectionSort (int array[], int size) {
    int minIndex;
    for (int index = 0; index < (size -1); index++) {
        minIndex = findIndexOfMin(array, size, index);
        swap(array[minIndex],array[index]);
    }
}
```
Bubble sort

On each pass:
- Compare first two elements. If the first is bigger, they exchange places (swap).
- Compare second and third elements. If second is bigger, exchange them.
- Repeat until last two elements of the list are compared.

Repeat this process until a pass completes with no exchanges.

Example

<table>
<thead>
<tr>
<th>Initial</th>
<th>After first swap</th>
<th>After second swap</th>
<th>After third swap</th>
<th>After fourth swap</th>
<th>After fifth swap</th>
<th>After sixth swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 2 3 8 9</td>
<td>2 &gt; 7, swap</td>
<td>2 &gt; 3, swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 7 3 8 9</td>
<td>7 &gt; 2, swap</td>
<td>7 &gt; 3, swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>!(7 &gt; 8), no swap</td>
<td>!(8 &gt; 9), no swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>9 &gt; 1, swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>finished pass 1, did 3 swaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: largest element is in last position

Example

<table>
<thead>
<tr>
<th>Initial</th>
<th>After first swap</th>
<th>After second swap</th>
<th>After third swap</th>
<th>After fourth swap</th>
<th>After fifth swap</th>
<th>After sixth swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 7 8 9</td>
<td>2 &lt; 3, no swap, !(3 &lt; 1), swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>2 &lt; 7, no swap, !(7 &lt; 1), swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>finished pass 2, did one swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 7 8 9</td>
<td>2 largest elements in last 2 positions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Initial</th>
<th>After first swap</th>
<th>After second swap</th>
<th>After third swap</th>
<th>After fourth swap</th>
<th>After fifth swap</th>
<th>After sixth swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 1 7 8 9</td>
<td>2 &lt; 3, no swap, !(3 &lt; 1), swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>2 &lt; 7, no swap, !(7 &lt; 1), swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>finished pass 3, did one swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>3 largest elements in last 3 positions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>finished pass 4, did one swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>finished pass 5, did one swap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 3 1 7 8 9</td>
<td>finished pass 6, no swaps, list is sorted!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Bubble sort**

**how does it work?**

- At the end of the first pass, the largest element is moved to the end (it’s bigger than all its neighbors)
- At the end of the second pass, the second largest element is moved to just before the last element.
- The back end (tail) of the list remains sorted.
- Each pass increases the size of the sorted portion.
- No exchanges implies each element is smaller than its next neighbor (so the list is sorted).

**Bubble sort: code**

```cpp
template<class ItemType>
void bubbleSort (ItemType a[], int size) {
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < (size-1); i++) {
            if (a[i] > a[i+1]) {
                swap(a[i],a[i+1]);
                swapped = true;
            }
        }
    } while (swapped);
}
```

**Quick sort**

- Divide and conquer!
- 2 (hopefully) half-sized lists sorted recursively
- the algorithm:
  - If list size is 0 or 1, return. otherwise:
    - partition into two lists:
      - pick one element as the pivot
      - put all elements less than pivot in first half
      - put all elements greater than pivot in second half
    - recursively sort first half and then second half of list.

**Quicksort Example.**

- Divide and conquer:
- 2 (hopefully) half-sized lists sorted recursively
- the algorithm:
  - If list size is 0 or 1, return. otherwise:
    - partition into two lists:
      - pick one element as the pivot
      - put all elements less than pivot in first half
      - put all elements greater than pivot in second half
    - recursively sort first half and then second half of list.
Quicksort: partitioning

- Goal: partition a sub-array so that:
- 4 8 5 6 3 19 12  pick some element as the pivot
- rearrange the array so that if the value is less than 6 it is placed before the 6, if the value is greater than the 6 it is placed after the 6.
- For an array, this might require some swapping and shifting.
- 4 3 5 6 8 19 12  return 3 as index of pivot (6)

Quicksort: code

```c
int partition (int [], int, int);  //defined in Gaddis

void quickSort(int array[], int start, int end) {
  if (start < end) {
    // Get the pivot point (and partition the set).
    int pivotPoint = partition(array, start, end);
    // Sort the first sub list.
    quickSort(array, start, pivotPoint - 1);
    // Sort the second sub list.
    quickSort(array, pivotPoint + 1, end);
  }
}

void quickSort (int array[], int size) {
  quickSort(array, 0, size-1);
}
```