Search & Sorting
Week 11
Gaddis: 8, 19.6, 19.8

CS 5301
Spring 2014
Jill Seaman

Definitions of Search and Sort

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

Linear Search

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

Linear Search in C++

```cpp
int searchList (int list[], int size, int value) {
    int index=0;          //index to process the array
    int position = -1;    //position of target
    bool found = false;   //flag, true when target is found

    while (index < size && !found)
    {
        if (list[index] == value)  //found the target!
        {
            found = true;            //set the flag
            position = index;        //record which item
        }
        index++;                   //increment loop index
    }
    return position;
}
```
Linear Search in C++

```cpp
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
  - See Lab 10, exercise #1: isMember function
- Linear search over linked list
  - A good exercise (Gaddis ch 17 Prog Challenge #5)
- 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
```

```
target < 50
```

```
target > 7
```

```
target == 11
```
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 {                          // search upper half
            first = middle + 1;
        }
    }
    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.

Sorting algorithms

- **Bubble sort**
- **Merge sort**
- **Quicksort**
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

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).

Example

7 2 3 8 9 1
7 > 2, swap

2 7 3 8 9 1
7 > 3, swap

2 3 7 8 9 1
!(7 > 8), no swap

2 3 7 8 9 1
!(8 > 9), no swap

2 3 7 8 1 9
9 > 1, swap

2 3 7 8 1 9 finished pass 1, did 3 swaps

Note: largest element is in last position

Example

2 3 7 8 1 9
2<3<7<8, no swap, !(8<1), swap

2 3 7 1 8 9
!(8<9) no swap

2 3 7 1 8 9 finished pass 2, did one swap

2 largest elements in last 2 positions

2 3 1 7 8 9
7<8<9, no swap

2 3 1 7 8 9 finished pass 3, did one swap

3 largest elements in last 3 positions
Bubble sort

Example

- 2 3 1 7 8 9  2<3, !(3<1) swap, 3<7<8<9
- 2 1 3 7 8 9
- finished pass 4, did one swap
- 2 1 3 7 8 9  !(2<1) swap, 2<3<7<8<9
- 1 2 3 7 8 9
- finished pass 5, did one swap
- 1 2 3 7 8 9  1<2<3<7<8<9, no swaps
- finished pass 6, no swaps, 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);
}
```

Merge sort

- Divide and conquer!
- 2 half-sized lists sorted recursively
- the algorithm:
  - if list size is 0 or 1, return (base case) otherwise:
  - recursively sort first half and then second half of list.
  - merge the two sorted halves into one sorted list.

Merge sort: Example

- Recursively divide list in half:
  - call mergeSort recursively on each one.
    - 5 2 4 6 1 3 2 6
    - 5 2 4 6 1 3 2 6
      - 5 2 4 6
      - 1 3 2 6
      - 5 2 4 6
      - 1 3 2 6
      - Each of these are sorted (base case length = 1)
Merge sort

Example

• Calls to merge, starting from the bottom:

  sorted sequence

  1 2 2 3 4 5 6 6

  merge

  2 4 5 6

  merge

  2 5

  merge

  4 6

  merge

  1 3

  merge

  2 6

  merge

  5 2

  merge

  4 6

  merge

  1 3

  merge

  2 6

  merge

  initial sequence

  1 2 2 3 4 5 6 6

Merge sort

Merging

• How to merge 2 (adjacent) lists:

  values

  1 13 24 26 2 15 27 38

  temp

  k

  first  middle  last

  1 13 24 26 2 15 27 38

  k

  1 2 13

  k

  compare values[i] to values[j], copy smaller to temp[k]

Merge sort

Merging

• Continued:

  values

  1 13 24 26 2 15 27 38

  temp

  k

  1 2 13 15 24

  k

  1 2 13 15 24 26

  k

  1 2 13 15 24 26 27

  k

  1 2 13 15 24 26 27

  k

  copy temp to values

Merge sort: code

void mergeSortRec (double values[], int first, int last) {
    if (first < last) {
        int middle = (first + last) / 2;

        mergeSortRec(values, first, middle);
        mergeSortRec(values, middle+1, last);

        merge(values, first, middle, last);
    }
}

void mergeSort (double values[], int size) {
    mergeSortRec(values, 0, size-1);
}
Merge sort: code: merge

```c
void merge(double values[], int first, int middle, int last) {
    double *tmp = new double[last-first+1]; //temporary array
    int i=first;        //index for left
    int j=middle+1;     //index for right
    int k=0;            //index for tmp

    while (i<=middle && j<=last)   //merge, compare next elem from each array
        if (values[i] < values[j])
            tmp[k++] = values[i++];
        else
            tmp[k++] = values[j++];

    while (i<=middle)           //merge remaining elements from left, if any
        tmp[k++] = values[i++];

    while (j<=last)             //merge remaining elements from right, if any
        tmp[k++] = values[j++];

    for (int i = first; i <=last; i++) //copy from tmp array back to values
        values[i] = tmp[i-first];

    delete [] tmp;                 //deallocate temp array
}
```

Quick sort

- Another 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

13 81 43 31 57 75 0

Quicksort

Example cont.

quick sort small

quick sort large

13 0 43 31 65 75 81 92

0 13 26 31 43 57 65 75 81 92
Quicksort: partitioning

- Goal: partition a sub-array A[start...end] by rearranging the elements and returning the index of the pivot point p so that:
- the algorithm:
  - pick a pivot value and swap with start elem
  - let pivotIndex = start and scan = start + 1

\[
\begin{array}{c}
4 & 8 & 5 & 6 & 3 & 19 & 12 \\
\end{array} \rightarrow
\begin{array}{c}
5 & 8 & 4 & 3 & 19 & 12 \\
\end{array}
\]

Note: pivotIndex is not always the midpoint

Quicksort: code

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

void quickSort(int set[], int size) {
    quickSort(set, 0, size-1);
}
```
int partition(int set[], int start, int end)
{
    int mid = (start + end) / 2;  // locate the pivot value
    swap(set[start], set[mid]);
    int pivotIndex = start;
    int pivotValue = set[start];
    for (int scan = start + 1; scan <= end; scan++)
    {
        // finds values less than pivotValue and
        // moves them to the left of the pivotIndex
        if (set[scan] < pivotValue)
        {
            pivotIndex++;
            swap(set[pivotIndex], set[scan]);
        }
    }
    swap(set[start], set[pivotIndex]);
    return pivotIndex;
}