Data Structures in C

Prof. Georg Feil

Sorting Algorithms

Winter 2018

Acknowledgement

- These lecture slides are based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- Read Data Structures (recommended textbook)
  - Chapter 1 sections 1.1 – 1.5
  - Chapter 2 sections 2.5
  - Chapter 10 (all sections)

Note the textbook does a few things we might consider poor style, for example one-letter variable names and using int for Boolean values.

Sorting Algorithms

- In this module we'll look at the following sorting algorithms (some of which you've seen before)
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Heap Sort
  - Quicksort
  - Shell Sort

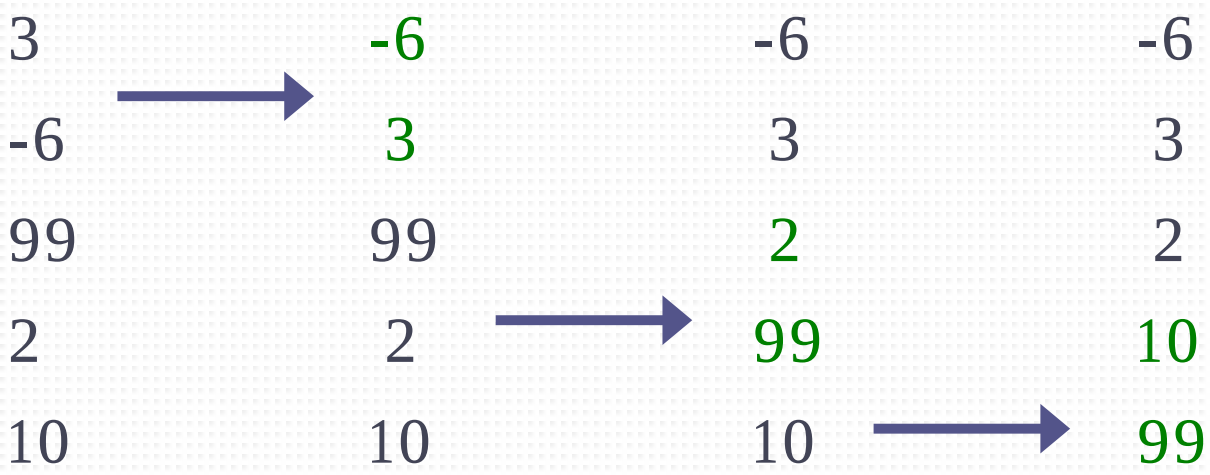
Sorting Algorithms

- These web sites help you visualize how the different sorting algorithms work
  - https://www.toptal.com/developers/sorting-algorithms/
  - http://www.bluffton.edu/homepages/facstaff/nesterd/java/SortingDemo.html
- Different algorithms are better suited to different types of sorted data
  - Data sets might be large or small. Data might be random, nearly sorted, totally reversed, or very similar in nature.
- Depending on the data, different algorithms are more or less useful – though they all do the same thing

Bubble Sort

- Bubble Sort is one of the easiest sorts to implement
- It iterates through the list continually swapping adjacent values if it finds they're out of order
  - After the first pass the largest value is at the end
- The sort stops after a pass where no swaps were required

Bubble Sort First "Pass"



Bubble Sort Code

- One possible implementation is shown below

```
bool swapped = true;  // Did we swap?
while (swapped) {
    swapped = false;
    for (int i = 0; i < n - 1; i++) {
        // Check if we need to swap
        if (a[i] > a[i+1]) {
            // Swap two array elements
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;

            swapped = true;  // Show that a swap occurred
        }
    }
}
```

Bubble Sort Complexity

- Bubble Sort is probably the worst sort in terms of performance
- It runs in $O(n^2)$ time
- It normally does a large number of swaps
- It's fast if the list is close to sorted – maybe $O(n)$

Selection Sort

- Like Bubble Sort, Selection Sort is an easy sort to implement
- It iterates through the list and finds the smallest number
  - Then, it moves the smallest number to the beginning
  - Then it iterates through the remaining list and finds the second smallest number
  - Then, it moves it to one from the beginning
- Repeat
- Equivalent sort: Each time find the largest number and move it to the end

Selection Sort Code

- One possible implementation is shown below

```
for (int j = 0; j < n-1; j++) {
    int iMin = j;
    for (int i = j+1; i < n; i++) {
        if (a[i] < a[iMin]) {
            iMin = i;
        }
    }

    if (iMin != j) {
        swap(&a[j], &a[iMin]);
    }
}
```

Selection Sort Complexity

- The performance of Selection Sort is almost as bad as Bubble Sort
- It always runs in $O(n^2)$ time, though it does the fewest number of swaps
- On the same data Selection Sort is usually faster than Bubble Sort (because it does fewer swaps), but the difference is not very significant
  → They are both $O(n^2)$!

Insertion Sort

- The next algorithm worth mentioning is Insertion Sort
- It works by taking each number in turn and inserting it into the partially sorted list in the correct place until the list is completely sorted
  - Remember our linked list exercise that did something similar?
- Essentially, it iterates through the list and inserts each number where it should go (a little like the opposite of a selection sort)
- The implementation on the next slide finds an element that's not in order, then moves the remaining elements ahead to make room

Insertion Sort Code

- One possible implementation is shown below

```
for (int i = 1; i < n; i++) {
    if (a[i] < a[i-1]) {
        temp = a[i];
        for (int j = i; j>0 && a[j-1]>temp; j--) {
            a[j] = a[j-1];
        }
        a[j] = temp;
    }
}
```

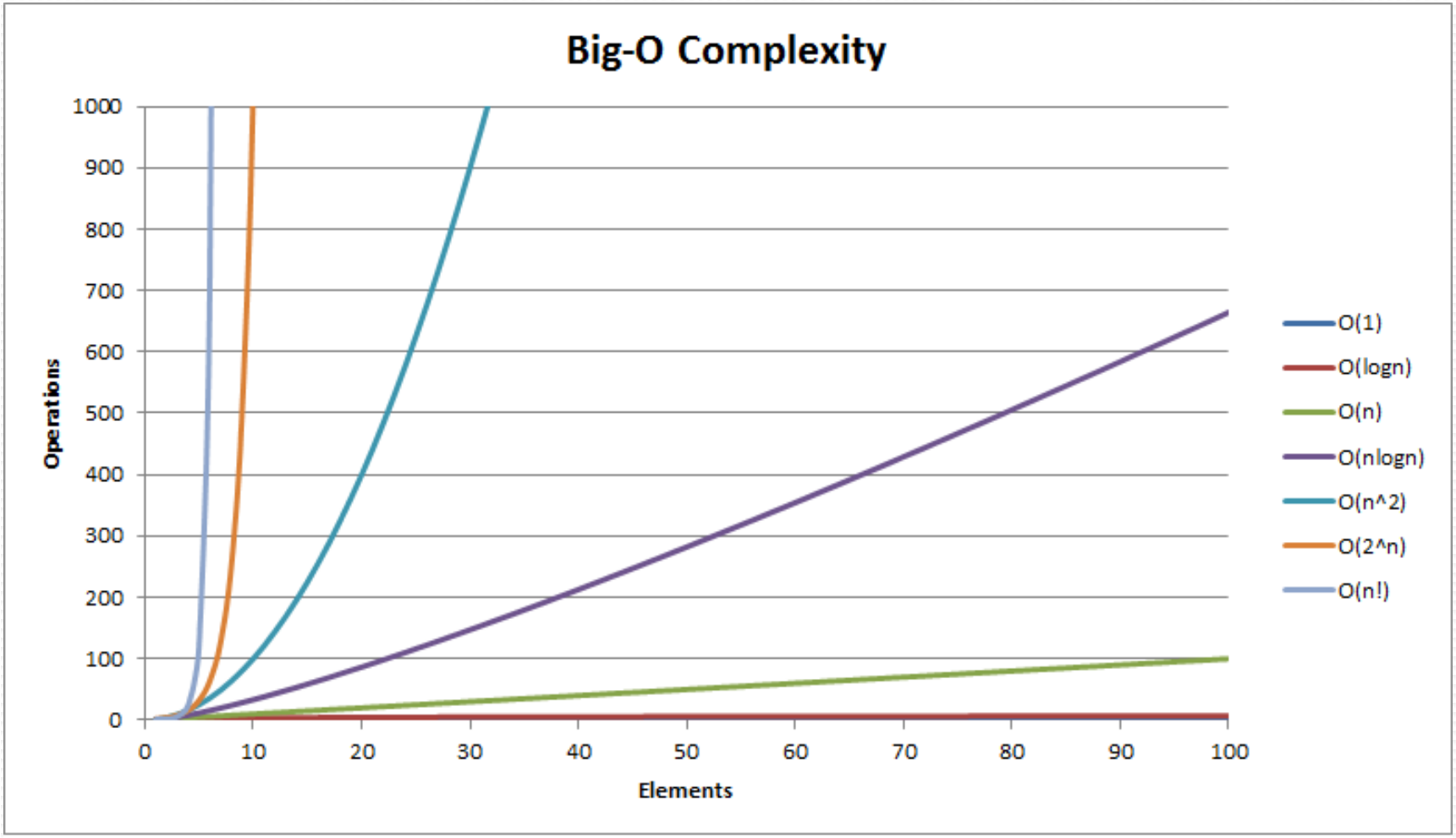
Insertion Sort Complexity

- Insertion Sort also runs in $O(n^2)$ worst-case time
- However, it is often a good choice if the data is nearly sorted!
  - It is sometimes used as a recursive base case (after most of the list has already been sorted) by higher divide-and-conquer algorithms like Quicksort or Merge Sort
- It's time is roughly $O(n)$ if the data is nearly sorted, which is actually not bad –but this almost never happens by itself

Advanced Sorting Algorithms

We can do much better than $O(n^2)$

Common Big O values (complexity classes)



Heap Sort

- Heap sort interprets the data as an almost complete binary tree
- More technically, a (max) heap is an almost complete binary tree such that the root is greater than or equal to the value at the left and right children, and the left and right children are also heaps
- Organize the data in a tree –no particular ordering –just start at the root, then the left child, then the right child, then the left-left child, then the left-right child, etc.
- This video may help to visualize Heap Sort https://www.youtube.com/watch?v=WYII2Oau_VY

Heap Sort Algorithm

- Starting at the last non-leaf node entered, convert the partial tree rooted there to a max-heap
  - May require a swap, may not, may require two checks or maybe just one
  - A max-heap has the largest number at the root
- Then go to the next non-leaf node to the left and make the partial tree there a max-heap
- When you reach the left side of the tree, start over a level up from the right again and repeat
  - Remember, each partial tree must be a smaller, but complete, max-heap – this may mean you have to go multiple levels down with re-heaps

Heap Sort Algorithm

If that wasn't tricky enough, here comes a hard part...

- Now move the node at the root (top) to the last node in the heap
  - Could also just remove it (e.g. move it to a list or queue) as in the video
- Re max-heap the new tree
- Place the new root in the next node to the left and re max-heap, etc.

Heap Sort Code

```
void HeapSort(int n) {
    for (int i = n/2; i>= 0; i--)
        ReHeap(n, i);
    for (int i = n-1; i> 0; i--) {
        swap(&a[i], &a[0]);
        ReHeap(i, 0);
    }
}
void ReHeap(int len, int parent) {
    temp = a[parent];
    int child = 2*parent + 1;
    while (child < len) {
        if (child<len-1 && a[child]<a[child+1])
            child++;
        if (temp >= a[child])
            break;
        a[parent] = a[child];
        parent = child;
        child = 2*parent + 1;
    }
    a[parent] = temp;
}
```

Heap Sort Complexity

- The first loop puts the data in heap order
- The second loop re-heaps the data repeatedly after extracting the maximum each time
- Heap sort is considered quite good, it runs in order O(n log(n))
- It is complicated though, and not often used
  - Strangely, in a nearly sorted case, heap sort destroys the original order
  - However, in a reversed case, heap sort is as fast as possible!

Heap Sort Complexity Comparison

- If we assume a computer can process one million comparisons per second, and we want to sort one million items, which sort is fastest so far?
- The selection sort will require $O(n^2) = 1{,}000{,}000^2$ comparisons, taking 1,000,000 seconds (about 12 days!)
- The heap sort will do it in $O(n \log_2(n)) = \sim 20{,}000{,}000$ comparisons or about 20 seconds!
- Number of comparisons:
  - Selection sort: `1,000,000,000,000`
  - Heap sort: `20,000,000`

Quicksort

- Quicksort involves partitioning the list with respect to a pivot
- Pick one of the numbers in the list and consider it a pivot
  - Possible choices: last number, middle number, random number
- Move the numbers so that all numbers smaller than the pivot come first, then the pivot, then all numbers larger than the pivot
- Split the list at the pivot's new position
- We now have two halves of the problem to sort, but we can use the same process to sort them, so recursion is used

Quicksort Code

```
void QuickSort(int low, int high) {
    if (high <= low) return;
    pivot = a[high];
    int split = low;
    for (int i=low; i<high; i++) {
        if (a[i] < pivot) {
            swap(&a[i], &a[split]);
            split++;
        }
    }
    swap(&a[high], &a[split]);
    QuickSort(low, split-1);
    QuickSort(split+1, high);
}
```

Quicksort Algorithm

- Look at each number in the array up to the pivot
  - If the number is bigger than the pivot value we do nothing
  - If it is smaller, we swap it with the number at the split position. Initially the split is 0, but it increases by 1 each time we swap
  - In this way we divide the numbers into two groups: Those that are smaller than the pivot first, followed by those that are larger
- One more swap of the pivot with the number at the split position puts the pivot value in the right place
- We then divide the list at the split position and quicksort each part recursively
- See http://www.youtube.com/watch?v=Fju_NstEy3M

Quicksort Complexity

- The computational complexity (performance) of Quicksort is a bit ambiguous
- Technically, it is O(n log(n)), but performance depends on what we choose as the pivot
  - If we choose the smallest number as the initial pivot, we essentially get a selection sort, $O(n^2)$
- There are several ways to "try" to avoid this (random pivot, median pivot, dual pivot etc.)
  - What works best depends on the properties of the data to be sorted

Shell Sort

- Shell Sort (named after Donald Shell) compares elements that are a certain number of steps apart, swapping if needed
  - Start with larger steps and moving to smaller ones
  - A good gap sequence is ..., 103, 46, 20, 9, 4, 1 (Tokuda)
- For example we may begin by comparing and swapping all elements that are 103 elements apart (gap of 103)
  - We compare index 0 and 103, 1 and 104, 2 and 105, etc.
  - Then we compare elements 46 apart and so on
- Shell Sort is not as fast as Heapsort or Quicksort but it's easier to implement and takes less memory

Shell Sort

- The idea is that after each incremental sort, the list is in a “closer-to-sorted” order
  - This means the next step in the sort takes less time
- The code given on the next slide uses a simple “not so good” gap sequence (outer for loop), not the better Tokuda sequence mentioned earlier
  - The first gap size is half the data size (for 16 items it would be 8), the next gap is half that and so on

Shell Sort Code

```
for (int inc = n/2; inc > 0; inc /= 2) {
    for (int i = inc; i < size; i++) {
        int j = i-inc;
        while (j >= 0) {
            if (a[j] > a[j+inc]) {
                swap(&a[j], &a[j+inc]);
                j -= inc;
            } else {
                j = -1;
            }
        }
    }
}
```

Shell Sort Complexity

- The computational complexity of Shell Sort depends on the gap sequence
- For the simple gap sequence used in the code it's $O(n^2)$
- For a better gap sequence like Tokuda's (..., 103, 46, 20, 9, 4, 1) the complexity is unknown but better than $O(n^{4/3})$
  - This is not as good as O(n log(n))
- For more information see https://en.wikipedia.org/wiki/Shellsort

Sorting Algorithms Summary

We looked at these sorting algorithms

- Bubble Sort: $O(n^2)$, requires many swaps (slow)
- Selection Sort: $O(n^2)$, requires few swaps
- Insertion Sort: $O(n^2)$, fast with nearly sorted list
- Heap Sort: $O(n \log(n))$ always, but complex
- Quicksort: $O(n \log(n))$ but sometimes $O(n^2)$, efficient
- Shell Sort: ~ $O(n^{4/3})$ with good gap sequence, needs very little memory

Which sorting algorithm to use?

- In general, if you don't know the nature of the data, or if it is random in nature, quicksort is a good choice
  - Use a good version of Quicksort that doesn't run into problems due to poor choice of pivot!
- If you know the nature of the data (list size, random, nearly sorted, almost reversed, or with few unique values/permutations), do some research and try to find the best algorithm
- With large data sets a good sorting algorithm can save a lot of of time, processing power, and memory, which in turn translates to saving money

Exercise 1

Suppose you need to write a sort algorithm for each of the situations described below. Which algorithm would be a good choice in each case?

1. Very large data set containing random values
2. Medium-size data set and your program needs to run on an embedded system (small computer, not much memory)
3. Large data set, values are nearly sorted
4. Very small data set

What programmers *actually* do

- Smart programmers avoid writing sorting algorithms if at all possible
  - The chance that your algorithm will have a bug is pretty high
- Use sorting functions supplied in your language library
- Java has Arrays.sort()
  - The Java documentation says:
    "The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations."
- C has qsort()
  - Implemented differently on different systems
  - Usually uses Quicksort with good way to choose pivot, e.g Cygwin
  - Linux uses a Merge Sort algorithm, code here

C library qsort function

- Function prototype for qsort:

```
void qsort(void* base, size_t nmemb, size_t size,
           int (*compar)(const void*, const void*));
```

  - base points to the start of an array
  - nmemb is the number of values in array
  - size is the size in bytes of each element in the array
  - compar is a *comparison function* that takes two parameters to compare and returns an int (it's actually a function pointer)
- `compar` parameter: C lets you pass function pointers as parameters to functions!
  - The closest thing in Java is Lambda expressions

The qsort comparison function

- When using qsort you must write a function that compares two of the values in your array, declared
  `int cmpfunc(const void* a, const void* b)`
- The comparison function must return an integer
  - -1 (or any other negative number) if the first parameter value is considered to be less than the second parameter value
  - 0 if the first parameter value is considered to be equal to the second parameter value
  - 1 (or any other positive number) if the first parameter value is considered to be greater than the second parameter value
- Pass the name of the function – no parentheses – as the last parameter to qsort

The qsort comparison function

- `int cmpfunc(const void* a, const void* b)`
- The parameters are flexible, could point to a value of any data type
  - int, double, structure etc.
- In cmpfunc you cast the void pointers to the correct data type
- The function should return
  - A positive number (say 1) if a should come after b, i.e. a > b
  - Zero if a == b
  - A negative number (say -1) if a should come before b, i.e. a < b

Example of using qsort() with int array

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 90, 88, 56, 100, 2, 25 };

int cmpfunc(const void* a, const void* b) {  // Integer comparison function
    return ( *(int*)a - *(int*)b );   // Why does this work?
}

int main(int argc, char** argv) {
    const int size = sizeof(values)/sizeof(int);  // # items in the array
    printf("Before sorting the list is: \n");
    for (int i = 0 ; i < size; i++) {
        printf("%d ", values[i]);
    }

    qsort(values, size, sizeof(int), cmpfunc);  // Sort the array "in place"

    printf("\nAfter sorting the list is: \n");
    for (int i = 0 ; i < size; i++) {
        printf("%d ", values[i]);
    }

    return 0;
}
```

Exercise 2

Write a C program that does the following:

1. Prompt the user to enter 5 floating-point numbers, then input the numbers and store them in an array.
2. Sort the numbers in descending order using qsort (not ascending order as in the example on the previous slide).
3. Print the sorted numbers.

Hint: Your comparison function won't be able to use the subtraction trick as with integers, change it to use good old 'if'!

Using qsort with an array of structures

- Because we have full control over the comparison function, and it uses void pointers, we can use qsort to sort any array... including arrays of structures
  - We can sort by any field of the structure (or any combination of fields), in either ascending or descending order
  - In the comparison function we cast the void pointer to the structure-pointer data type, then dereference with -> to access the desired field(s)

- See the program qsortStruct.c in the same SLATE folder as these slides for an example

Sorting a linked list

Sorting a linked list

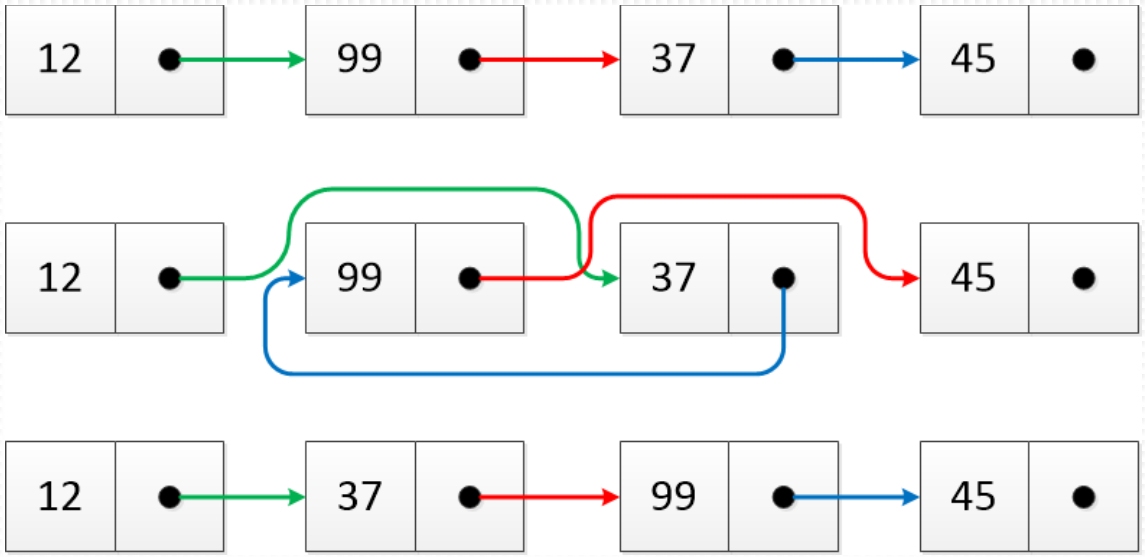
- Suppose you have a linked list of 'Student' structures

```
typedef struct student {
    char name[NAME_STR_LEN];
    long int ID;
    double GPA;
    struct student* next; // Next student towards tail
} Student;
```

- How can we sort this linked list without using an array?
  - Sort key could be name, ID, or GPA

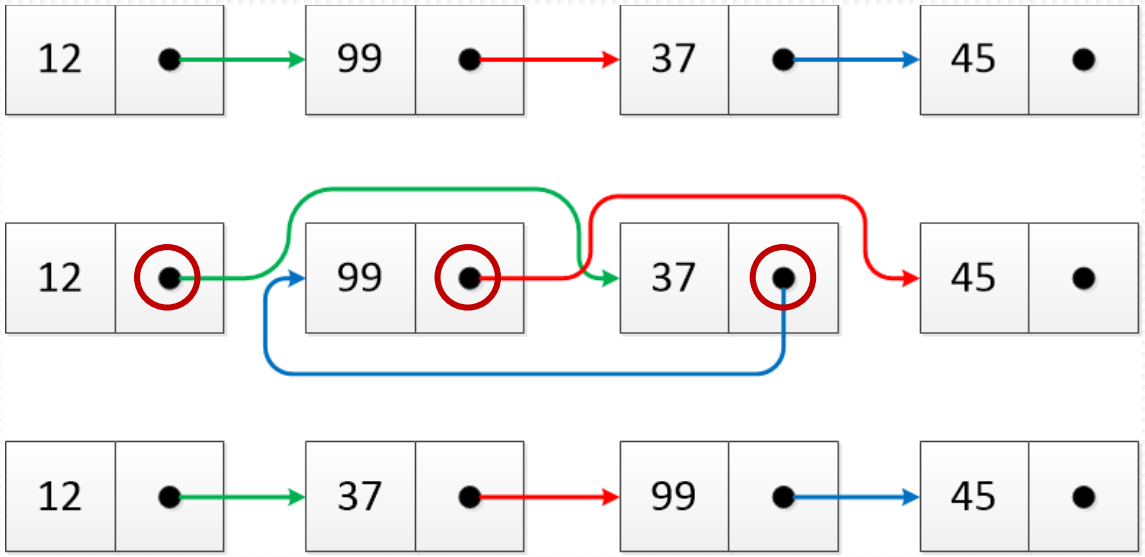
Sorting a linked list: Bubble Sort

- For example, consider a sort that requires swapping elements like Bubble Sort
- We could adjust next pointers to swap adjacent nodes



Sorting a linked list: Bubble Sort

- Swapping requires changing three next pointers, and possibly the head or tail pointer
  - In general we need to keep track of recently visited nodes or *traverse* the list to find nodes and change their next pointers



Sorting a linked list: Bubble Sort!

```
void bsort(List* list) {
    bool swapped = true;  // Did we swap?
    while (swapped) {
        swapped = false;
        Student* last = NULL;  // 'st' last time around inner loop
        Student* st = list->head;
        while (st != NULL && st->next != NULL) {  // Traverse the list
            // Check if we need to swap
            if (st->ID > st->next->ID) {
                // Swap two adjacent list elements
                Student* nxx = st->next->next;  // Node two ahead
                if (last != NULL)
                    last->next = st->next;  // (green)
                else
                    list->head = st->next;  // Update head if needed

                if (list->tail == st->next)
                    list->tail = st;  // Update tail if needed

                st->next->next = st;  // (blue)
                Student* tmp = st->next;  // Save for below
                st->next = nxx;  // (red)

                st = tmp;        // Set 'st' to new correct node
                swapped = true;  // Show that a swap occurred
            }
            last = st;      // Remember last value of 'st'
            st = st->next;  // Advance to the next node
        }
    }
}
```

Sorting a linked list: Other sorts

- Sorting linked data structures is efficient when the size of each node structure is large
  - You don't have to copy the node contents as with an array, just change next pointers
- Other sort algorithms are possible, but would be even more complicated