# Data Structures

# Today's Agenda

- **Sorting Algorithms: Recursive**
  - mergesort
  - quicksort
- As we learn about each sorting algorithm, we will discuss its efficiency
- Review for the Final Exam

# Mergesort

- The mergesort is considered to be a divide and conquer sorting algorithm (as is the quicksort ).
- The mergesort is a recursive approach which is very efficient.
- The mergesort can work on arrays, linked lists, or even external files.
- At first glance, it doesn’t seem like a sorting algorithm at all...

# Mergesort

- The mergesort is a recursive sorting algorithm that always gives the same performance regardless of the initial order of the data.
  - For example, you might divide an array in half - sort each half - then merge the sorted halves into 1 data structure.
  - To merge, you compare 1 element in 1 half of the list to an element in the other half, moving the smaller item into the new data structure.

# Mergesort

- The sorting method for each half is done by a recursive call to merge sort.
  - That is why this is a divide and conquer method.
- Mergesort(list,starting place, ending place)
  - if the starting place is less than the ending place

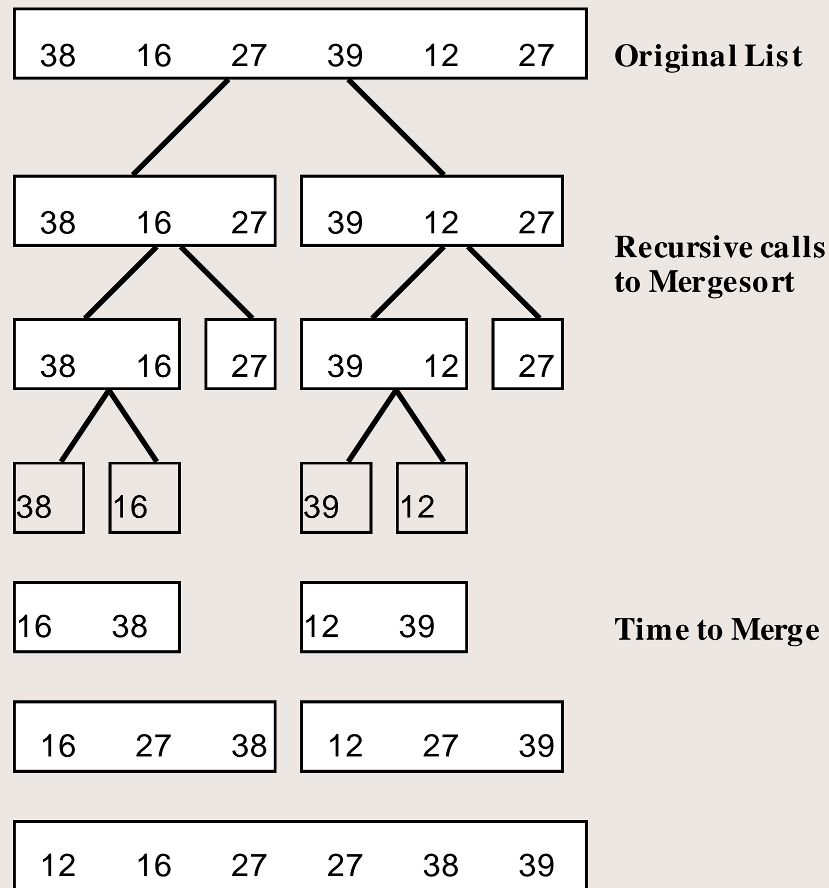
    middle place = (starting + ending) div 2

    mergesort(list, starting place, middle place)

    mergesort(list,middle place+1, ending place)

    merge the 2 halves of the list

# Mergesort




**Original List**

**Recursive calls to Mergesort**

**Time to Merge**

# Mergesort

- If we implemented this approach using arrays ---
  - If the total number of items in your list is m...then for each merge we must do m-1 comparisons.
  - For example, if there are 6 items we must do five comparisons.
  - In addition, there are m moves from the original location to some temporary location (and back).

# Mergesort

- Even though this seems like a lot, you will see that this is actually faster than either the selection sort or the insertion sort.
- Although the mergesort is extremely efficient with respect to time, it does require that an equal "temporary" array be used which is the same size as the original array.
  - If temporary arrays are not used...this approach ends up being no better than any of the others

# Mergesort

- If we implement the mergesort using linked lists, we do not need to be concerned with the amount of time needed to move data
  - Instead, we just need to concentrate on the number of comparisons.
  - When lists get very long, the number of comparisons is far less with the mergesort than it is with the insertion sort.
  - Problems requiring 1/2 hour of computer time using the insertion sort will probably require no more than a minute using the mergesort.

# Mergesort

- Remember, the mergesort is a recursive sorting algorithm
  - that always gives the same performance regardless of the initial order of the data.
  - For example, you might divide an array in half - sort each half - then merge the sorted halves into 1 data structure.
  - To merge, you compare 1 element in 1 half of the list to an element in the other half, moving the smaller item into the new data structure.

# Mergesort

- Start by looking at the merge operation.
  - Each merge step merges your list in half.
  - If the total number of elements in the two segments to be merged is m then merging the segments requires m-1 comparisons.
  - In addition, there are m moves from the original array to the temporary array and m moves back from the temporary array to the original array.
  - 3*m-1 major operations in the merge step

# Mergesort

- Now we need to remember that each call to mergesort calls itself twice.
  - How many levels of recursion are there?
  - Remember we continue halving the list of numbers until the result is a piece with only 1 number in it.
  - Therefore, if there are N items in your list, there will be either $\log_2 N$ levels (if N is a power of 2) and $1+\log_2 N$ (if N is NOT a power of 2).

# Mergesort

- Remember, each one of these calls requires 3*M-1 operations, where M starts equal to N, then becomes N/2.
- When M = N/2, there are actually 2 calls to merge, so there are 2*(3*M-1) operations or 6(N/2)-2 =>>>>> 3N-2.
- Expanding on this: at recursive call m, there are 2m calls to re-merge where each call merges $N/2^m$ elements, so it requires $3*(N/2^m)-1$ operations.

# Mergesort

- Which is the same as $3*N-2^m$ .
- Using the BIG O approach, this breaks down to O(N) operations at each level of recursion.
- Because there are either $\log_2 N$ or $1+\log_2 N$ levels, mergesort altogether has a worst and average case efficiency of $O(N*\log_2 N)$

# Mergesort

- If you work through how log works, you will see that this is actually significantly faster than an efficiency of $O(N^2)$.
- Therefore, this is an extremely efficient alg.
- The only drawback is that it requires a temporary array of equal size to the original array.
  - This could be too restrictive when storage is limited.

# Quicksort

- The quicksort is also considered to be a divide and conquer sorting algorithm.
- The quicksort partitions a list of data items that are less than a pivot point and those that are greater than or equal to the pivot point.
- You could think of this as recursive steps:
  - step 1 - choose a pivot point in the list of items
  - step 2 - partition the elements around the pivot

# Quicksort

- This generates two smaller sorting problems, sorting the left section of the list and the right section (excluding the pivot point...as it is already in the correct sorted place).
- Once each of the left and right smaller lists are sorted in the same way, the entire list will be sorted (this terminates the recursive algorithm).

# Quicksort

- Notice that partitioning the list is probably the most difficult part of the algorithm.
- It must arrange the elements in two regions: those greater than the pivot and those less.
  - The first question which might come to mind is which pivot to use?
  - If the elements are arranged randomly, you can chose a pivot randomly.

# Quicksort

- We are not required to choose the first item in the list as the pivot point.
  - We can choose any item we want and swap it with the first before beginning the sequence of partitions.
  - In fact, the first item is usually not a good choice...many times the first item in a list is already sorted.
  - That would mean that there would be no items in the left partition!

# Quicksort

- Therefore, it might be better to chose a pivot from the center of the list, hoping that it will divide the list approximately in two.
  - If you are sorting a list that is almost in sorted order...this would require less data movement!
  - At each step in the partition function, we need to examine one element in the unknown region, determine how it relates to the pivot point, and place it in one of the two regions (< or =>).
  - think of this as making piles...

# Quicksort

- Notice that quicksort actually alters the array itself...not a temporary array.
- Quicksort and mergesort are very similar.
  - Quicksort does work before its recursive calls.
  - Mergesort does work after its recursive calls.
- Sort in class the following using both methods:
  - 29, 10, 14, 37, 13, 12, 30, 20

# Quicksort

- The worst case with this method is when one of the regions (left or right) remains empty (i.e., the pivot value was the largest or smallest of the unknown region).
- This means that one of the regions will only decrease in size by only 1 number (the pivot) for each recursive call to Quicksort.

# Quicksort

- Also, notice what would happen if our array is already sorted in ascending order?
- If we pick a pivot value as the first value, for each recursive call we only decrease the size by 1 -- and do SIZE-1 comparisons.
- Therefore, we will have many unnecessary comparisons.

# Quicksort

- The good news is that if the list is already sorted in ascending order and the pivot is the smallest #, then we actually do not perform any moves.
- But, if the list is sorted in descending order and the pivot is the largest, there will not only be a large number of un-necessary comparisons but also the same number of moves.

# Quicksort

- On average, quicksort runs much faster than the insertion sort on large arrays.
- In the worst case, quicksort will require roughly the same amount of time as the insertion sort.
  - If the data is in a random order, the quicksort performs at least as well as any known sorting algorithm that involves comparisons.
  - Therefore, unless the array is already ordered -- the quicksort is the best bet!

# Quicksort

- Mergesort,  on the other hand,
  - runs somewhere between the Quicksort's best and worst case (insertion sort).
  - Sometimes quicksort is faster; sometimes it is slower!
  - The thing to keep in mind is that the worst case behavior of the mergesort is about the same as the quicksort's average case.

# Quicksort

- Usually the quicksort will run faster than the mergesort...
  - but if you are dealing with already sorted or mostly sorted data, you will get worst case performance out of the quicksort which will be significantly slower than the mergesort.
  - with an array that is already sorted in ascending order and the pivot is always the smallest element in the array, then there are N-1 comparisons for N elements in your array.

# Quicksort

- On the next recursive call there are N-2 comparisons, etc.
  - This will continue leaving the left hand side empty doing comparisons until the size of the unsorted area is only 1.
  - Therefore, there are N-1 levels of recursion and 1+2+...+(N-1) comparisons...which is: N*(N-1)/2.
  - The good news that in this case there are no exchanges.

# Quicksort

- Also, if you are dealing with an array that is already sorted in descending order and the pivot is always the largest element in the array, there are N*(N-1)/2 comparisons and N*(N-1)/2 exchanges (requiring 3 data moves each).
- Therefore, we should be able to quickly remember that this is just like the insertion sort -- and in the worst case has an efficiency of $O(N^2)$.

# Quicksort

- Now look at the case where we have a randomly ordered list and pick reasonably good pivot values that divide the list almost equally in two.
  - This will require fewer recursive calls (either $\log_2 N$ or $1+\log_2 N$ recursive calls).
  - Each call requires M comparisons and at most M exchanges, where M is the number of elements in the unsorted area and is less than N-1.

# Quicksort

- We come to the realization that in an average-case behavior, quicksort has an efficiency of $O(N*\log_2 N)$.
  - This means that on large arrays, expect Quicksort to run significantly faster than the insertion sort.
  - Quicksort is important to learn because its average case is far better than its worst case behavior -- and in practice it is usually very fast.

# Quicksort

- It can out perform the mergesort if good pivot values are selected.
- However, in its worst case, it will run significantly slower than the mergesort (but doesn't require the extra memory overhead).

# Review for the Final Exam

- **The Final Exam in CS163 is**
  - comprehensive
  - closed book
  - closed notes
- It will focus on:
  - trees (BST, 2-3, AVL, 2-3-4, red-black, heaps)
  - graphs (depth first, breadth first traversal)
  - sorting algorithms (insertion, selection, mergesort, quicksort)

# Review for the Final Exam

- In addition, you will be asked questions about:
  - hash tables using chaining
  - stacks, queues, ordered lists
  - various data structures such as LLL, circular linked lists, doubly linked lists, doubly threaded lists, arrays of linked lists, linked lists of arrays

# Review for the Final Exam

- When implementing a table abstract data type, explain why you would select:
  - a) hash table
  - b) binary search tree
  - c) red-black tree
  - d) 2-3 tree
  - e) graph
  - f) linear linked list
  - g) 2-3-4 tree
  - h) doubly linked list

# Review for the Final Exam

- Given the following data, draw the following trees: 40 20 25 60 65 70 73 15
  - AVL tree
  - 2-3 tree
  - BST tree
  - Heap
  - 2-3-4 tree
  - red-black tree

# Review for the Final Exam

- Now, delete 40 from
  - AVL tree
  - 2-3 tree
- Now, delete 65 from
  - AVL tree
  - 2-3 tree

# Review for the Final Exam

- Given the following data, show the steps of sorting: 40 20 25 60 65 70 73 15
  - using insertion sort
  - using selection sort
  - using exchange sort
  - using mergesort
  - using quicksort
  - discuss the efficiency of each...which is better and why

# Review for the Final Exam

- Explain the process of inserting data into a 2-3 tree.
- Explain the process of deleting a node from a BST
  - what special cases do we need to consider
  - how do we delete a node that has two children? (write the algorithm)