Projects

Data structures

1. Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers. What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

2. Implement a city database using list. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. Another operation that should be supported is to print all records within a given distance of a specified point. Implement the database using an array-based list implementation

3. Write a recursive function named **smallcount** that, given the pointer to the root of a BST and a key K, returns the number of nodes having key values less than or equal to K. Function **smallcount** should visit as few nodes in the BST as possible.

4. Write linked implementation for the deque. Initial value of deque elements are read from file. Deque must include the following fuctions: adding, and removing elements from both the front and the rear, listing elements, checking is deque full or empty.

5. Implement two stacks sharing the same array. Stacks must include the following fuctions: adding, and removing elements, listing elements, checking is stack full or empty. Find the maximum and minimum elements of the stack and swap them (do not brake LIFO principle).

6. Implement the Lithuanian-English dictionary using list data structure. Function **insert** takes a record and inserts it into the dictionary. Function **find** takes a word and returns some record from the dictionary whose word matches the one provided. If there are multiple records in the dictionary with that word value, there is no requirement as to which one is returned. The **remove** function is similar to find, except that it also deletes the record returned from the dictionary. Function **size** returns the number of elements in the dictionary.

7. Write a program that includes recursive function that returns the height of a binary tree and count of the number of leaf nodes in the tree. Also write a function that takes as input the pointer to the root of a binary tree and prints the node values of the tree in level order. Level order first prints the root, then all nodes of level 1, then all nodes of level 2, and so on.

8. Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers. What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

9. Implement a city database using list. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x and y coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. Another operation that should be supported is to print all records within a given distance of a specified point. Implement the database using an array-based list implementation

10. Write a recursive function named **smallcount** that, given the pointer to the root of a BST and a key K, returns the number of nodes having key values less than or equal to K. Function **smallcount** should visit as few nodes in the BST as possible.

11. Write linked implementation for the deque. Initial value of deque elements are read from file. Deque must include the following fuctions: adding, and removing elements from both the front and the rear, listing elements, checking is deque full or empty.

12. Implement two stacks sharing the same array. Stacks must include the following fuctions: adding, and removing elements, listing elements, checking is stack full or empty. Find the maximum and minimum elements of the stack and swap them (do not brake LIFO principle).

13. Implement the Lithuanian-English dictionary using list data structure. Function **insert** takes a record and inserts it into the dictionary. Function **find** takes a word and returns some record from the dictionary whose word matches the one provided. If there are multiple records in the dictionary with that word value, there is no requirement as to which one is returned. The **remove** function is similar to find, except that it also deletes the record returned from the dictionary. Function **size** returns the number of elements in the dictionary.

14. Write a program that includes recursive function that returns the height of a binary tree and count of the number of leaf nodes in the tree. Also write a function that takes as input the pointer to the root of a binary tree and prints the node values of the tree in level order. Level order first prints the root, then all nodes of level 1, then all nodes of level 2, and so on.

15. Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers. What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

Sorting algorithms

1. One possible improvement for Bubble Sort would be to add a flag variable and a test that determines if an exchange was made during the current iteration. If no exchange was made, then the list is sorted and so the algorithm can stop early. Modify the Bubble Sort implementation to add this flag and test. Compare the modified implementation on a range of inputs to determine if it does or does not improve performance in practice.

2. Perform a study of Shellsort, using different increments. Compare the version shown in Theory slides, where each increment is half the previous one, with others. In particular, try implementing "division by 3" where the increments on a list of length n will be n/3, n/9, etc. Do other increment schemes work as well?

3. The implementation for Mergesort given Theory slides takes an array as input and sorts that array. There is a simple pseudocode implementation for sorting a linked list using Mergesort. Implement both a linked list-based version of Mergesort and the array-based version of Mergesort, and compare their running times.

4. Write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.
   (Look at more values when selecting a pivot).

5. Perform a study of Shellsort, using different increments. Compare ShellSort version where each increment is half the previous one, with others. In particular, try implementing "division by 3" where the increments on a list of length n will be n/3, n/9, etc. Do other increment schemes work as well?

6. Modify Quicksort to sort a sequence of variable-length strings stored one after the other in a character array, with a second array (storing pointers to strings) used to index the strings. Your function should modify the index array so that the first pointer points to the beginning of the lowest valued string, and so on.

7. Write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.
   (Look at more values when selecting a pivot).

8. One possible improvement for Bubble Sort would be to add a flag variable and a test that determines if an exchange was made during the current iteration. If no exchange was made, then the list is sorted and so the algorithm can stop early. Modify the Bubble Sort implementation to add this flag and test. Compare the modified implementation on a range of inputs to determine if it does or does not improve performance in practice.

9. Perform a study of Shellsort, using different increments. Compare the version shown in Theory slides, where each increment is half the previous one, with others. In particular, try implementing "division by 3" where the increments on a list of length n will be n/3, n/9, etc. Do other increment schemes work as well?

10. The implementation for Mergesort given Theory slides takes an array as input and sorts that array. There is a simple pseudocode implementation for sorting a linked list using Mergesort. Implement both a linked list-based version of Mergesort and the array-based version of Mergesort, and compare their running times.

11. Write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.
    (Look at more values when selecting a pivot).

12. Perform a study of Shellsort, using different increments. Compare ShellSort version where each increment is half the previous one, with others. In particular, try implementing "division by 3" where the increments on a list of length n will be n/3, n/9, etc. Do other increment schemes work as well?

13. Modify Quicksort to sort a sequence of variable-length strings stored one after the other in a character array, with a second array (storing pointers to strings) used to index the strings. Your function should modify the index array so that the first pointer points to the beginning of the lowest valued string, and so on.

14. Write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.
    (Look at more values when selecting a pivot).

15. One possible improvement for Bubble Sort would be to add a flag variable and a test that determines if an exchange was made during the current iteration. If no exchange was made, then the list is sorted and so the algorithm can stop early. Modify the Bubble Sort implementation to add this flag and test. Compare the modified implementation on a range of inputs to determine if it does or does not improve performance in practice.