

## Problems

1. (50 pts) Implement a hash for text. Given a string as input, construct a hash with words as keys, and word counts as values. Your implementation should include:
  - a hash function that has good properties for text
  - storage and collision management using linked lists
  - operations: insert(key,value), delete(key), increase(key), find(key), list-all-keys
  - each word (key) can only appear once in the datastructure

Output the list of words together with their counts on an output file. For this problem, you cannot use built-in-language datastructures that can index by strings (like hash tables or dictionaries). Use a language that easily implements linked lists.

You can test your code on “Alice in Wonderland” by Lewis Carroll, at [http://www.ccs.neu.edu/home/vip/teach/Algorithms/\7\\_hash\\_RBtree\\_simpleDS/hw\\_hash\\_RBtree/alice\\_in\\_wonderland.txt](http://www.ccs.neu.edu/home/vip/teach/Algorithms/\7_hash_RBtree_simpleDS/hw_hash_RBtree/alice_in_wonderland.txt).

The test file used by TA, if different, might be shorter.

Try these three values for  $m = \text{MAXHASH}$  : 30, 300, 1000. For each of these  $m$  values, produce a histogram over the lengths of collision lists. You can also calculate variance of these lengths.

If your hash is close to uniform in collisions, you should get variance close to zero, and almost all list-lengths around  $\alpha = n/m$

If your hash has long lists, we want to know how many and how long, for example print the lengths of the longest 10% of the lists.

(Extra Credit) Find a way to record not only word counts, but also the positions in text. For each word, besides the count value, built a linked list with positions in the given text. Output this list together with the count.

2. (50points) Implement a red-black tree, including binary-search-tree operations *sort*, *search*, *min*, *max*, *successor*, *predecessor* and specific red-black procedures *rotation*, *insert*, *delete*.  
You must implement *delete* with functionality for regular BST (without fixup procedure) but implementing *delete* for RB-trees (including fixup procedure) is Extra Credit.  
Your code should take the input array of numbers from a file and build a red-black tree with this input by sequence of “inserts”. Then interactively ask the user for an

operational command like “insert x” or “sort” or “search x” etc, on each of which your code rearranges the tree and if needed produces an output. After each operation also print out the height of the tree.

You can use any mechanism to implement the tree, for example with pointers and struct objects in C++, or with arrays of indices that represent links between parent and children. You cannot use any tree built-in structure in any language.

3. ( Implement Skiplists 50 points). Study the skiplist data structure and operations. They are used for sorting values, but in a datastructure more efficient than lists or arrays, and more guaranteed than binary search trees. Review Slides [skiplists.pdf](#) and Visualizer <https://people.ok.ubc.ca/y1ucet/DS/SkipList.html>. The demo will be a sequence of operations (asked by TA) such as for example insert 20, insert 40, insert 10, insert 20, insert 5, insert 80, delete 20, insert 100, insert 20, insert 30, delete 5, insert 50, lookup 80, etc

4. (50 pts). Implement binomial heaps as described in class and in the book. You should use links (pointers) to implement the structure as shown in the figure 1. Your implementation should include the operations: Make-heap, Insert, Minimum, Extract-Min, Union, Decrease-Key, Delete

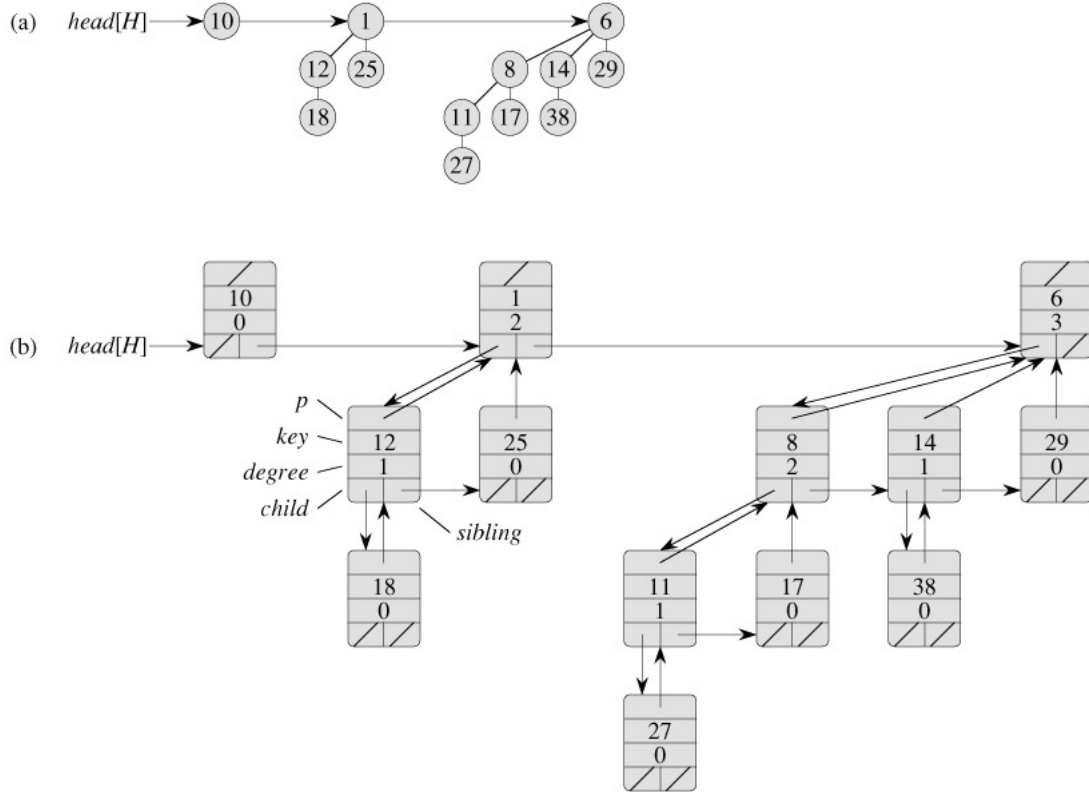


Figure 1: binomial heaps

Make sure to preserve the characteristics of binomial heaps at all times: (1) each component should be a binomial tree with children-keys bigger than the parent-key; (2) the binomial trees should be in order of size from left to right. Test your code several arrays set of random generated integers (keys).

5. Exercise 16.3-3



Input 6: {1, 1, 2, 2, 4, 4, 8, 8, 16, 16, 32, 32, 64, 64, 128, 128, 256, 256, 512, 512, 1024, 1024, 2048, 2048, 4096, 1, 1, 2, 2, 4, 4, 8, 8, 16, 16, 32, 32, 64, 64, 128, 128, 256, 256, 512, 512, 1024, 1024, 2048, 2048, 4096}

Return 6: 62648678

Input 7: {10000, 9999, 9998, 9997, 9996, 9995, 9994, 9993, 9992, 9991, 9990, 9989, 9988, 9987, 9986, 9985, 9984, 9983, 9982, 9981, 9980, 9979, 9978, 9977, 9976, 9975, 9974, 9973, 9972, 9971, 9970, 9969, 9968, 9967, 9966, 9965, 9964, 9963, 9962, 9961, 9960, 9959, 9958, 9957, 9956, 9955, 9954, 9953, 9952, 9951}

Return 7: 89655114688016

Input 8: {9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999}

Return 8: 4814999

Input 9: {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 10000}

Return 9: 6555