Java Data Structures Replacement

Songnian Yin, syin2@stevens.edu
Yabin Han, yhan14@stevens.edu
Ying Cui, ycui11@stevens.edu

ArrayList in Integer and Double

- Constructor:
expand()
size()
isEmpty()
toArray()
- Modification Operations:
add()
remove()
clear()
trimToSize()
- Positional Access Operations:
get()
set()
indexOf()
subList()
- Iterator

LinkedList in Integer and Double

- Node class
- Constructor
size()
isEmpty()
toArray()
- Modification Operations:
add()
addFirst()
addLast()
remove()
clear()
doClear()
- Positional Access Operations:
getValue()
getFirst()
getLast()
subList()
- Iterator

Trie

- N-nary Tree
- Prefix Tree
- Share same element
- Dictionary Implement
- Level order Store

Hash Function Analysis

- Murmur
- SuperfastHash
- DJB2
- FNV-1a

ArrayList<Integer/Double>

We use an array of integer and double to implement the arraylist of integer and double. In our program, there are mainly four part functions. Constructor, modification operations, positional access operations and iterator.

We use size to represent the real size of arraylist and capacity to represent the volume of the arraylist. When the size is larger than capacity, we make the capacity doubly which is the expand function in our program.

In the modification operations part, we have function add() which means add element in the last of array, remove() which means delete the first element in the array, clear() which clears all the elements in the arraylist, trimToSize() which makes the capacity to equal to the array size.

In the positional access operations part, get() function means get the elements of input index. The set() function sets the input element in the input index. There are also add() and remove() functions which adds or remove the element in the specific position. The sublist() function is to get the sublist of this arraylist from the begin index to end index.

In the iterator part, we override three functions. The hasNext(), next() and remove. We make a currentIndex points to the array[0]. If the arraylist has the next element, the currentIndex should smaller than size. So we return the boolean result of the inequality of currentIndex and size. If it has the next element, return the element whose index is currentIndex+1. The remove function removes the element on currentIndex and makes the currentIndex-1. We use the remove() function in modification operations again.

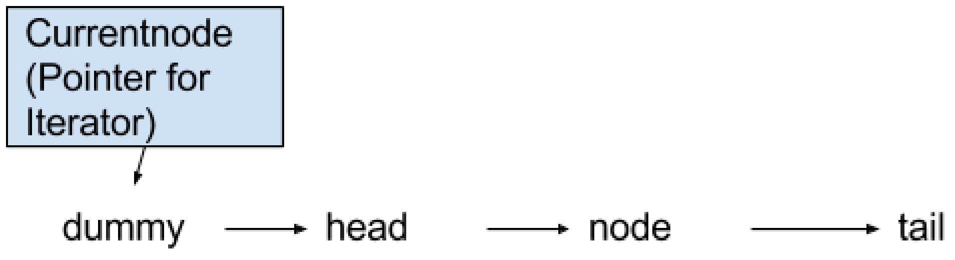
LinkedList <Integer/Double>

We build double linkedlist with two pointer: previous and next. And we also has two node head and tail as pointer. Since we use the Integer and Double Object which bulid in the java API, we did not beat the java API when we create our linkedlist. This object use lots of memory. It waste lots of time when we create it. So we think maybe we can use primitive variable such as int or double or to rewrite the Integer or Double class. I think we will try it in the future.

Another thing we learned in this project is that the dummy node is very useful in lots of situation. Such as in our project we need to use the dummy node to connect the head when we build the Iterator.

Dummy node:

http://www.cs.uwm.edu/faculty/boyland/classes-archive/fa15.cs351/www/linked-list-variations.html
Doubly linked list:
http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_linklist.aspx



Trie

The trie is a kind of special data structure which looks like the tree but it has more properties. The difference between them is the node structure.

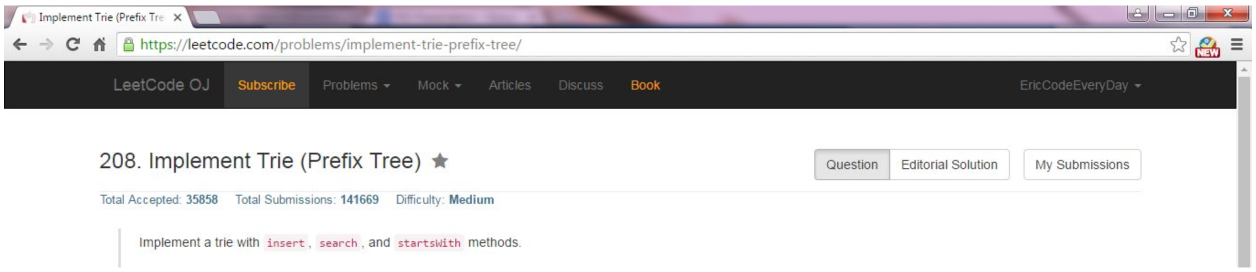
We consider the trie is n-nary tree and it can share the prefix. Like apple and apples, they share the prefix “apple”. The trie can save the memory space because of the share of prefix.

And another thing is we save the trie with BFS algorithm, in other words, the level order traversal. Then we save the trie into the disk system.

If we want to search the trie or update the trie, we can just load the trie from disk system then do what we want. Finally, we can save the trie back to the disk system.

When we want to code with Chinese word, we read some article from the following website:

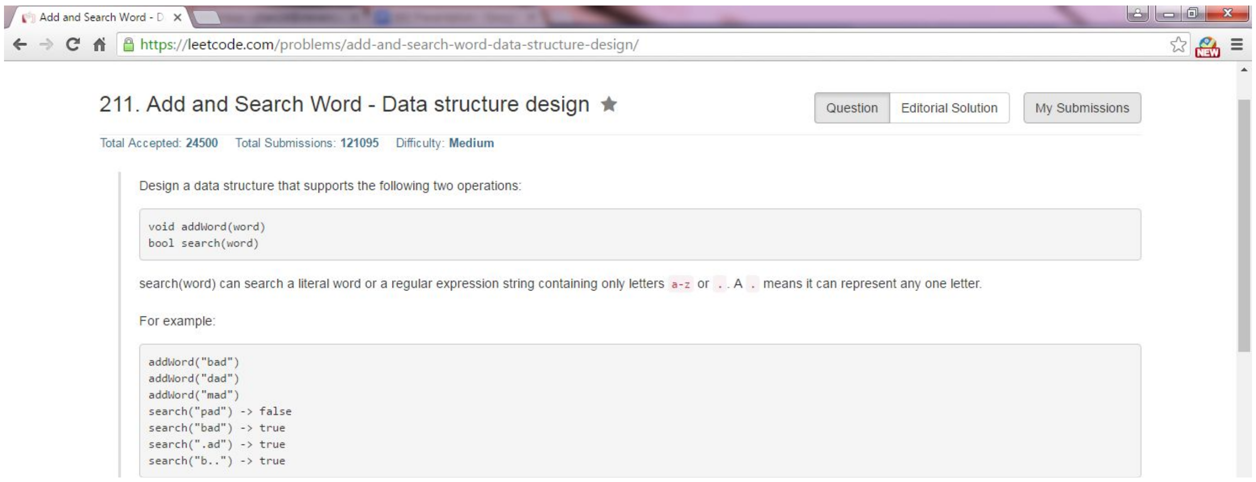
https://en.wikipedia.org/wiki/Trie
http://www.unicode.org/charts/
https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/
Some idea from leetcode as following



208. Implement Trie (Prefix Tree)

Total Accepted: 35858 Total Submissions: 141669 Difficulty: Medium

Implement a trie with `insert`, `search`, and `startsWith` methods.


Add and Search Word - D
https://leetcode.com/problems/add-and-search-word-data-structure-design/
Question
Editorial Solution
My Submissions

211. Add and Search Word - Data structure design

Total Accepted: 24500 Total Submissions: 121095 Difficulty: Medium

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

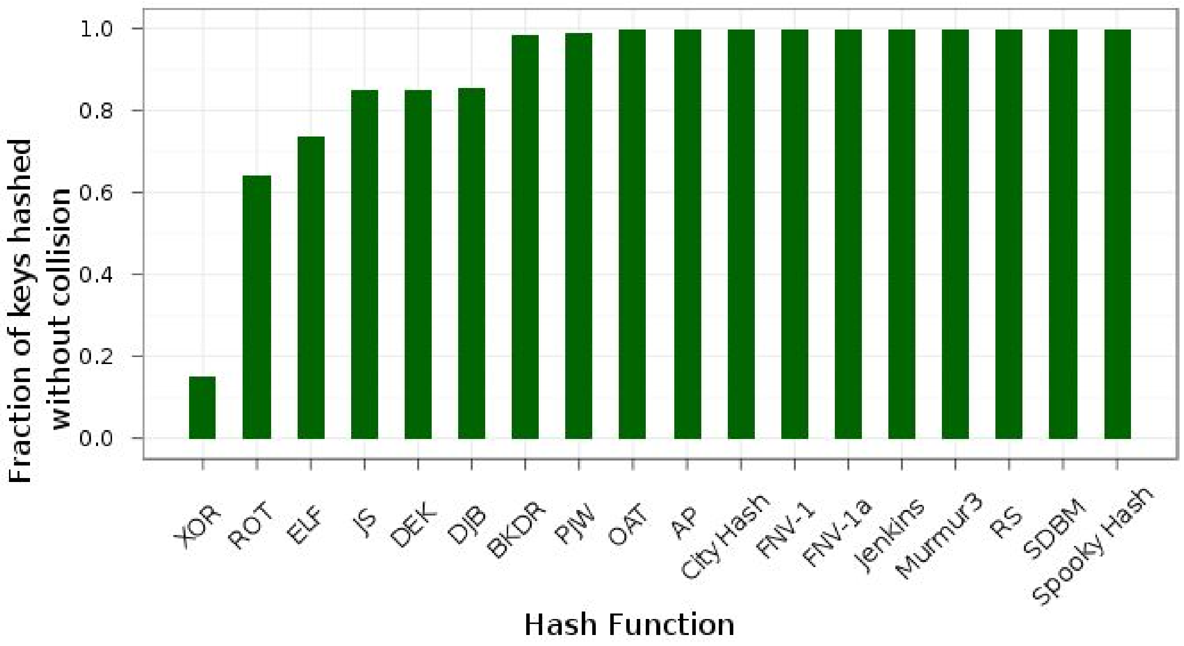
search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.`. A `.` means it can represent any one letter.

For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Hash Function

At first, we find a list of hash functions from the following website https://en.wikipedia.org/wiki/List_of_hash_functions



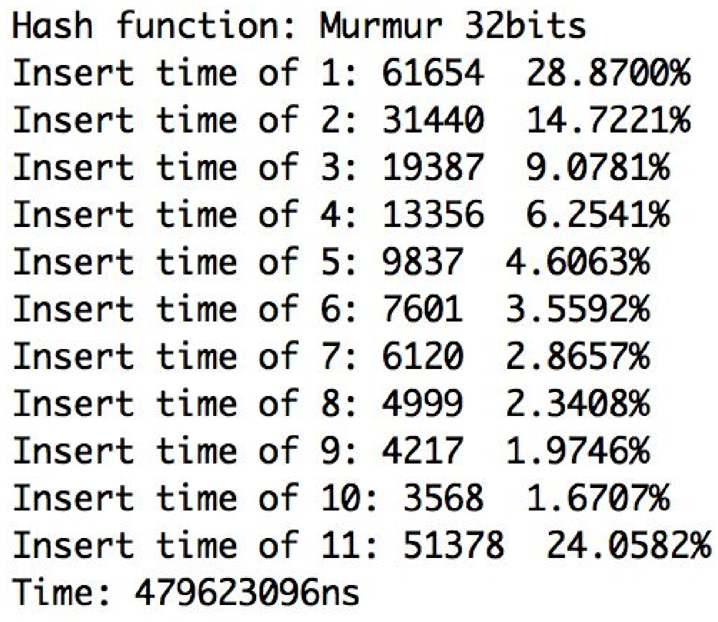
We finally implement several representative hash functions whose performance is outstanding on the efficiency and collision.

All the hash function are tested by the same dictionary.

Murmur3(used in java API)

https://en.wikipedia.org/wiki/MurmurHash

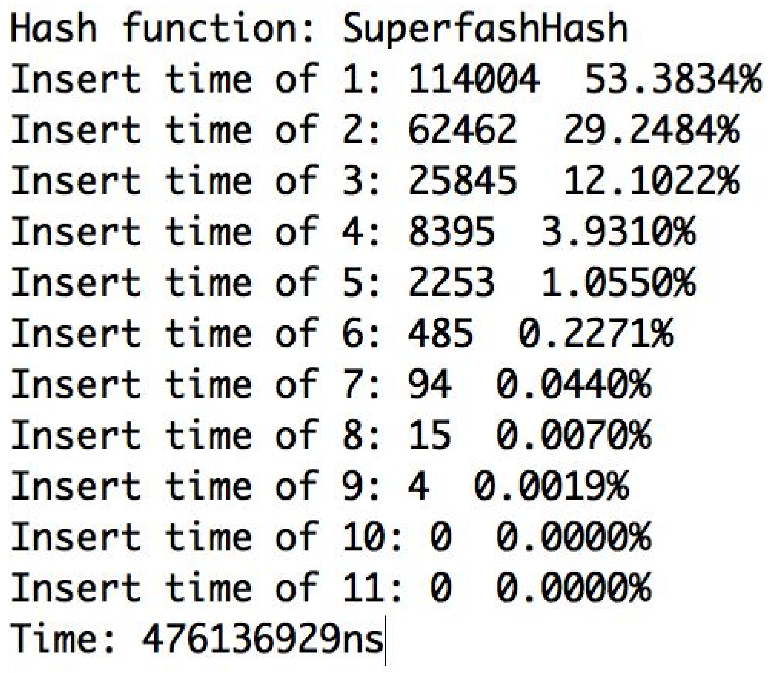
And the following is our result of implement.



```
Hash function: Murmur 32bits
Insert time of 1: 61654  28.8700%
Insert time of 2: 31440  14.7221%
Insert time of 3: 19387  9.0781%
Insert time of 4: 13356  6.2541%
Insert time of 5: 9837  4.6063%
Insert time of 6: 7601  3.5592%
Insert time of 7: 6120  2.8657%
Insert time of 8: 4999  2.3408%
Insert time of 9: 4217  1.9746%
Insert time of 10: 3568  1.6707%
Insert time of 11: 51378  24.0582%
Time: 479623096ns
```

SuperfastHash

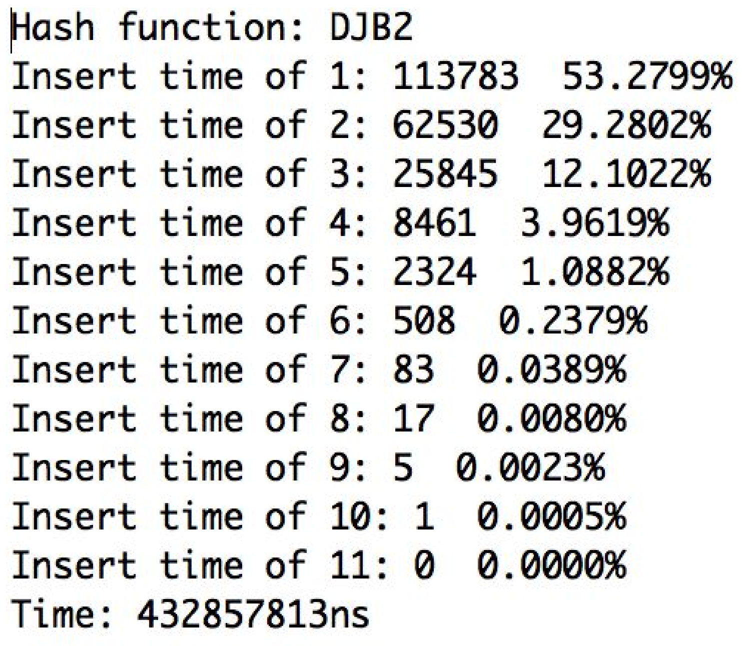
http://www.azillionmonkeys.com/qed/hash.html



```
Hash function: SuperfashHash
Insert time of 1: 114004  53.3834%
Insert time of 2: 62462  29.2484%
Insert time of 3: 25845  12.1022%
Insert time of 4: 8395  3.9310%
Insert time of 5: 2253  1.0550%
Insert time of 6: 485  0.2271%
Insert time of 7: 94  0.0440%
Insert time of 8: 15  0.0070%
Insert time of 9: 4  0.0019%
Insert time of 10: 0  0.0000%
Insert time of 11: 0  0.0000%
Time: 476136929ns
```

DJB2

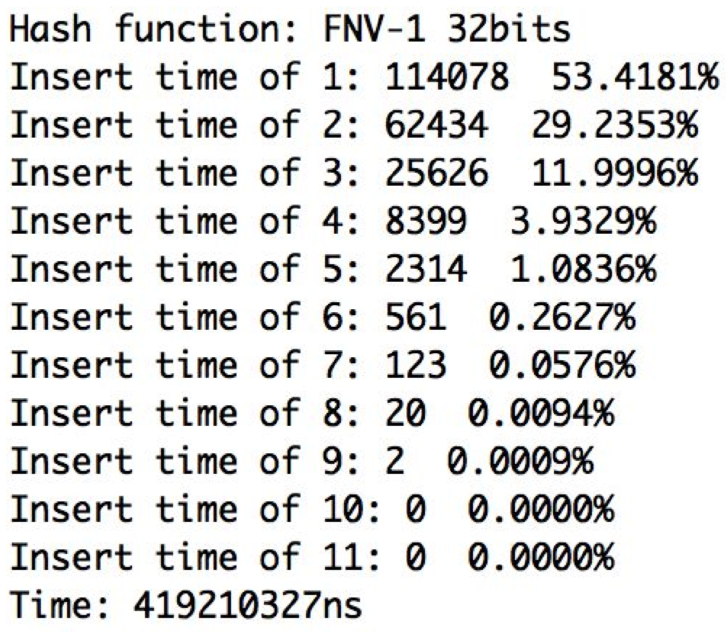
http://stackoverflow.com/questions/10696223/reason-for-5381-number-in-djb-hash-function/13809282#13809282



```
Hash function: DJB2
Insert time of 1: 113783  53.2799%
Insert time of 2: 62530  29.2802%
Insert time of 3: 25845  12.1022%
Insert time of 4: 8461  3.9619%
Insert time of 5: 2324  1.0882%
Insert time of 6: 508  0.2379%
Insert time of 7: 83  0.0389%
Insert time of 8: 17  0.0080%
Insert time of 9: 5  0.0023%
Insert time of 10: 1  0.0005%
Insert time of 11: 0  0.0000%
Time: 432857813ns
```

FNV-1a/FNV-1

https://en.wikipedia.org/wiki/Fowler–Noll–Vo_hash_function
http://www.isthe.com/chongo/tech/comp/fnv/
We implement the FNV-1a in 32bits. The hash function has two important number, FNV offset basis which we choose 0x811c9dc5 in this program and FNV prime which is 0x0100019.
The difference between FNV-1a and FNV-1 is the order of multiply and XOR. After testing, we choose FNV-1a to implement since it has better performance.



```
Hash function: FNV-1 32bits
Insert time of 1: 114078  53.4181%
Insert time of 2: 62434  29.2353%
Insert time of 3: 25626  11.9996%
Insert time of 4: 8399  3.9329%
Insert time of 5: 2314  1.0836%
Insert time of 6: 561  0.2627%
Insert time of 7: 123  0.0576%
Insert time of 8: 20  0.0094%
Insert time of 9: 2  0.0009%
Insert time of 10: 0  0.0000%
Insert time of 11: 0  0.0000%
Time: 419210327ns
```

Conclusion:

FNV-1a 32bits is the best hash function we find in our project. It has least collisions and costs shortest time.