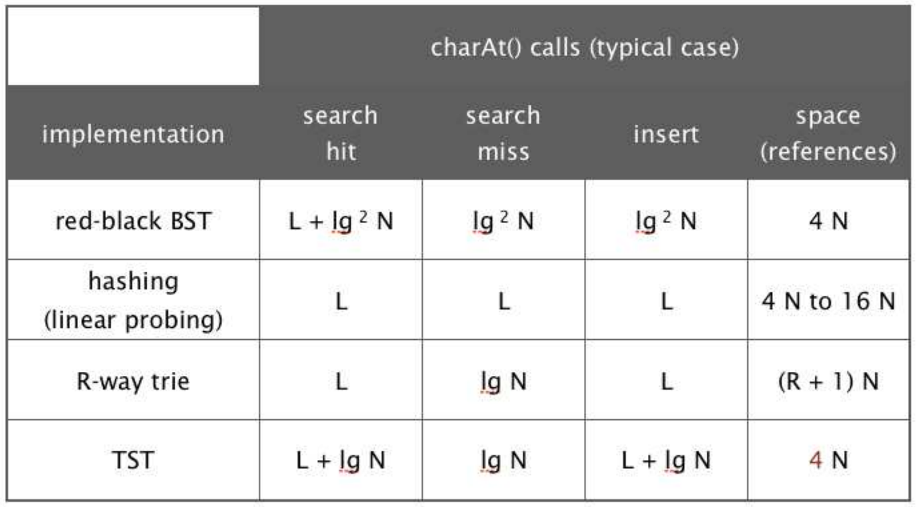
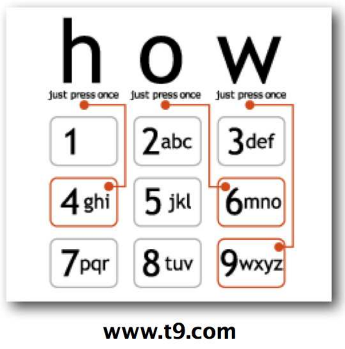
# COS226 Week 9 Group Activity Soultion



| | charAt() calls (typical case) | | | |
|---|---|---|---|---|
| implementation | search hit | search miss | insert | space (references) |
| red-black BST | L + lg $^2$ N | lg $^2$ N | lg $^2$ N | 4 N |
| hashing (linear probing) | L | L | L | 4 N to 16 N |
| R-way trie | L | lg N | L | (R + 1) N |
| TST | L + lg N | lg N | L + lg N | 4 N |

The values in this table only apply for a very specific string model and model of computation. They should be used as a rough guide only! Do not memorize or write on a cheat sheet. What matters is which entries are linear and which are sublinear.



1. Tries

   (a) Design problem

      i. Sequences of decimal digits, e.g. ”7447”.

      ii. A set of words, e.g. ”SHIV”, ”PIGT” Even better is an ordered set of word where more common words go up top.

      iii. An R-way trie, where the alphabet is the numbers 2 through 9. R-way tries will probably outperform TSTs in this case since R is mall, but experiments would be needed to validate this.

      iv.

   (b) The alphabet would have been the standard ASCII alphabet. One could also optimize a bit and include only english letters, but the ASCII alphabet would be fine. We’d expect the performance to be a bit better than an LLRB or a hash table. Again, experiments would be needed to validate this. In this case, we’d probably use a TST instead of an R-way trie.

   (c) In this case, we’d need to make a choice about the alphabet. Natural choices include binary digits and hex digits. At any rate, it’s certainly a very awkward thing to do.

   Interesting followup question: What data structure does our Trie devolve into if we chose an alphabet where R = 4 billion (i.e. all ints are just a single character in our alphabet).

   (d) Support for character based operations, for example keysWithPrefix. This is the main reason we care about tries!

   (e) Short answer: When the data isn’t a string.

   Using tries on things that aren’t really strings is very awkward. Getting access to a compact string representation of a complex object is a nontrivial task. For example, imagine trying to build a trie of Picture objects. It can be done, but it will be awkward.

Another example is if we have long strings and want to minimize the number of memory accesses. With a trie, we have to follow W links, which might require going to hard-drive or even offsite storage, causing lots of latency. With a hash table, we compute a single hash value and have to follow only a constant number of links (e.g. usually less than 5 or so if we resize our hash table so that the number of items is never more than 5 times the number of buckets).

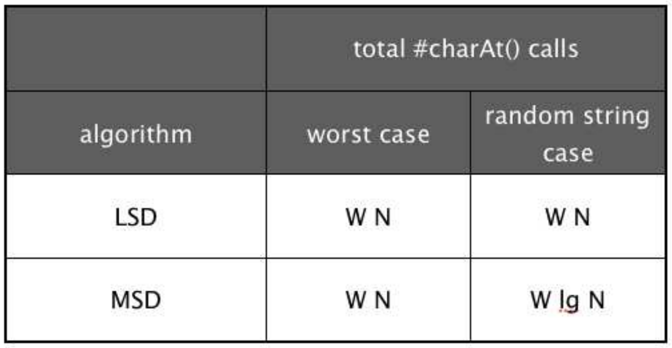
2. Sorting

   (a) MSD will use fewer compares. For example, the O in HORSE is never considered.

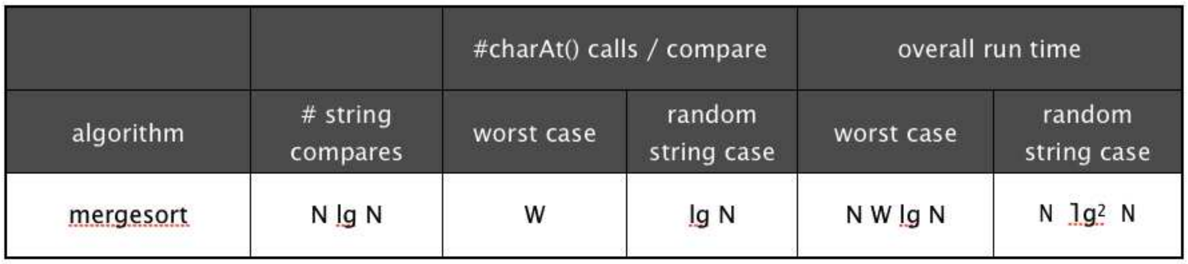
```
ZORSE    ZORSE -> ZERGS            BLERG    BLERG
HORSE    HORSE -> BLERG            BLARG    BLARG
BLERG    BLERG -> BLARG            HORSE -> HORSE
BLARG    BLARG -> ZORSE            ZORSE    ZERGS
ZERGS    ZERGS -> HORSE            ZERGS    ZORSE

orig     LSD 1    LSD 2            MSD 1    MSD 2
```

   (b) Fill in the tables below.



| | total #charAt() calls | |
|---|---|---|
| algorithm | worst case | random string case |
| LSD | W N | W N |
| MSD | W N | W lg N |



| | | #charAt() calls / compare | | overall run time | |
|---|---|---|---|---|---|
| algorithm | # string compares | worst case | random string case | worst case | random string case |
| mergesort | N lg N | W | lg N | N W lg N | N $\lg^2$ N |

   (c) MSD and mergesort like radically different strings, particularly in the most significant digit. They both perform particularly poorly (relative to their best case) on all-equal strings. For mergesort this is NOT because it performs extra string compares, but because each string compare takes longer!

   (d) For all equal strings, we expect LSD to perform better. This is because MSD has to go and create a bunch of extraneous count arrays. For very long strings (say megabytes), MSD is going to use a huge amount of memory.

**1.** Legume grime pop.

a. A hash table `originals` that maps strings to Bags of strings.

Also acceptable: any symbol table with string keys that can be constructed in time linear in the number of strings, e.g., an R-way trie. A red-black tree or TST would take $O(N \log N)$ time to construct.

b. Key idea: avoid generating every permutation of each word (which is $O(L!)$). **We can do this by creating a canonical representation for each anagram, namely the sorted version of the word.**

- Build a symbol table mapping the sorted version of each word to a bag of all words that sort to that same string. We take each word `word` from input, insertion sort it to generate `dorw`, and add `dorw` to the Bag `originals[w]` (create the Bag if it doesn't exist). Keep track of the maximum Bag length. $O(NL^2)$.
- Iterate through `originals` and print the max.

Note: Sorting each word using **key-indexed counting is asymptotically** faster at $O(NLR)$, where $R$ is the alphabet size, but **insertion sort** which is $O(NL^2)$ **is likely faster in practice** for typical English words.

c. Key idea: avoid an exponential search for all possible word ladders. Here are two solutions with varying tradeoffs between simplicity, speed and generalizability.

Both solutions use a subroutine `neighbors(dorw)` that assumes `originals` already exists – Given a dorw of length $k$, generate the $k$ dorws of length $k$-$1$ and return the ones that are valid (i.e., those that are keys in `originals`). $O(L^2)$.

Solution 1. **(Josh approved solution)**

- Create a DAG where nodes are dorws and there is an edge from each dorw to each of its neighbors. $N$ invocations of `neighbors`, $O(NL^2)$.
- Find the longest path in this DAG.
  - This can be done by creating a virtual root node with an edge to every root (finding the roots is $O(N)$), assigning a weight of -1 to each edge, and computing the *shortest* paths from the virtual root using topological sort. $O(N)$.
- Look up the dorws on this longest path (in reverse) in `originals` and print a sequence of original words.

Solution 2. **Less elegant but still perfectly fine solution.**

- Initialize a hash table `rung_height` from dorws to `ints` with all values set to *0*. The `rung_height` of `dorw` is the max rung height of `dorw` in any anagram ladder that it can appear in. Lowest rung is *0*.
- Create a sorted array of dorws in increasing order of length. $O(N)$ by key-indexed counting.
- For each dorw `dorw` In this array:

```
    rung_height[dorw] = max(rung_height[nbr] for nbr in
neighbors[dorw])
```

  // if `neighbors[dorw]` is empty do nothing, as `dorw` must be the bottom rung in any ladder. Note that the neighbors have already been processed because of sortedness.

  $N$ invocations of `neighbors`, $O(NL^2)$.
- Find the dorw with maximum `rung_height`, and iteratively find a sequence of neighbors with `rung_height` of each neighbor one less than the previous. $O(N + poly(L))$.
- Look up the dorws in this ladder (in reverse) in `originals` and print a sequence of original words.

Solution 1 is more elegant is but probably slower in practice (and consumes more memory) due to graph creation, despite having the same asymptotic runtime as Solution 2.