SPP PROJECT REPORT

2019121002 Aryamaan Jain
2019121001 Vipul Chhabra

Why we shifted from trie

- Submitted trie in submission-1. Had to shift from trie to b-tree because of huge memory consumption of trie. On runs of 10 million entries, out of memory issues took place in trie. Maximum number of entries trie supported were about 1.8 million on a 8GB laptop.
- Tested versions of trie which took less memory. On such version was TST(trenary search trie). Even TST was not able to meet memory requirements and gave hardly any memory improvements.

How TST failed

- **We ran a program to check the memory pattern of TST.**

```
 aryam@comp:~/Desktop/tst$ time ./a.out 1500000
num_calls: 91496643
sizeof node: 32
real    0m15.887s
user    0m13.286s
sys 0m2.566s
```

The program inserts 1.5 million key-values in TST.
Here the command line argument is number of key-values to be inserted.
num_calls is the number of calls for a new node.

- **Analysis of above output; Memory taken by program calculation**

    1. top command shows size of program ~ 3.9 GB
    2. num_calls * sizeof_node = 2927892576 bytes ~ 2.7268 GB
    3. back envolop calc = 64(key_len) * 32(sizeof_node) * (1.5 * 10^6)(N) = 3072000000 bytes ~ 2.861 GB

- **Inference**

    1. Worst case new_nodes = 64 * (1.5 * 10^6) = 96000000
    2. Observed new_nodes = num_calls = 91496643
    3. Which is hardly any advantage, the ratio being 1.0492188221594096.

- **Conclusion**
  The real advantage of trie comes in real world applications where words are not random and prefixes are much more common. There we can exploit the trie property of common matching prefix.
  In our case, where the benchmark is random, there is not much matching prefixes and so, there doesn't seem much of an advantage of using tries!

Comparision of various data structures

These comparsions are just for insertion, it may be completely possible that for rest of the 4 operations, the rankings may change completely. Example: AVL is faster than RB in searches.
These test cases are for insertions of 10 million integers in 8GB laptop and the codes were taken from random sources.

- **B-tree**
  With min_degree = 3

```
aryam@comp:~/Desktop/spp_proj/b$ time ./a.out 10000000

real	0m16.853s
user	0m16.332s
sys	0m0.483s
```

With min_degree = 16

```
aryam@comp:~/Desktop/spp_proj/b$ time ./a.out 10000000

real	0m10.314s
user	0m9.992s
sys	0m0.299s
```

- **AVL tree**

```
aryam@comp:~/Desktop/spp_proj/avl$ time ./a.out 10000000

real	0m49.381s
user	0m48.575s
sys	0m0.707s
```

- **RB tree**

```
aryam@comp:~/Desktop/spp_proj/rb$ time ./a.out 10000000

real	0m23.164s
user	0m22.644s
sys	0m0.475s
```

- **B+ tree** With numberOfPointers = 16

```
 aryam@comp:~/Desktop/spp_proj/bplus$ time ./a.out 10000000

real    0m19.916s
user    0m18.877s
sys 0m0.994s
```

Implementation choice: b-tree

Various factors affected out choice of b-tree. Some are listed below.

1. The above comparision shows that for insertions, b-tree is the fastest.
2. Stackoverflow: B-tree vs red-black or AVL Tree.
3. B-tree seemed easier to implement than red-black tree.
4. Cache hit / locality of b-tree is better than others since it stores values in arrays and the size of array can be optimised to fit cache line.

Optimisations applied in b-tree

1. For O(lg n) get(N) and del(N), we modified the structure to store number of children along each edge.
2. Optimised value of T(min-degree) for fitting cache line.
3. Further small optimisations we applied like if-else reordering for minimising branch misses, etc but not much difference in performance was observed.

Closing remarks

An unoptimised but good choice data structure beats an optimised but bad choice data structure!