Project Report

In-Memory Key-Value Storage

Team YEET

Istasis Mishra - 2018111028

Shradha Sehgal - 2018101071

Manvith Reddy - 201801057

Shanmukh Karra - 2018101111

Objective

Build an In-Memory Key-Value Storage Software in C++ that supports the APIs get(key), put(key, value), delete(key), get(int N), delete(int N).

Constraints

1. 10^7 entries
2. Max key size: 64 bytes
3. Chars can be between a-z or A-Z
4. Max value size: 256 bytes

Introduction

A key-value store, or key-value database is a simple database that uses some data structure to store keys and values such that each key is associated with one and only one value in a collection. This relationship is referred to as a **key-value pair.**

In each key-value pair the key is represented by an arbitrary string such as a filename, URI or hash. The value can be any kind of data like an image, user preference file or document. In our project we store strings for keys and integers for values.

We have to implement such a key-value store keeping in mind the TPS, CPU usage and memory usage have to be optimized by minimizing the formula **TPS*TPS/(CPU*Mem)** for a given system architecture.

The main challenge has been to choose a data structure support **both key access and lexicographical access**; in this document, we show how we overcame this and many other difficulties.

Baseline data structure

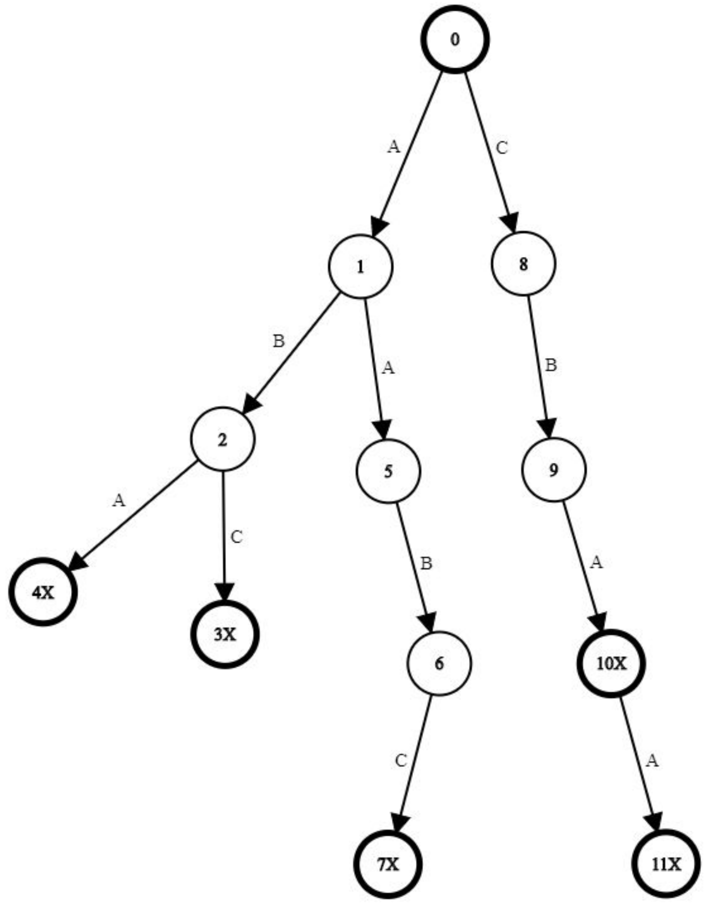
Tries are a commonly used data-structure for building key-value stores and indexes. They have several appealing properties – for example,

- Deals well with strings and uses less memory,
- Search for your key depends on the length of the key and not the number of nodes in the trie,
- Height of a tree is independent of the number of keys it contains,
- Requires no rebalancing when updated.

However, a trie by itself is a naïve approach, and there are multiple more efficient methods, distinguished by the number of children the node actually has, and not just how many it might potentially have. This is where **BST** comes in.

The problem with naive tries is that the span of the trie for our project is 52. For our project, we are restricted to 52 possible characters, which means the span of the trie is 52. The **trade-off between the span and the height of the tree** is a critical parameter. Moreover, **wasted space in each node** is a huge concern.

Initially, a completely untouched trie with the keys {AABC, ABA, ABC, CBA, CBAA} looks something like this:



The nodes with “X” beside them represent if the current node is the end of a string.

Challenges and Optimizations

1. Lexicographical accesses

For this functionality, we had to augment the nodes with another variable. Now the struct looks like this:

```
struct TrieNode
{
   int arr[53];
   bool end = false;
   int ends = 0;

};
```

The variable “ends” signifies the number of leaves (or number of word endings) in the current node’s subtree and bool ”end” signifies whether a word ends at the node. Doing this recursively, we find the **Nth lexicographical** subtree in **O(52L)** time, where L is the length of the string. However, this is the worst case scenario and on average the kvStore will perform much better as the keys are randomly generated. Therefore, hardly any access will take as long.

2. Free Lists

We used free lists to **avoid dynamic memory allocation** to increase TPS. Instead of creating new nodes or deleting old nodes, we **reuse the already created nodes** by maintaining a linked list of the free nodes.

We have used this in multiple locations. An example is shown here:

```
if (nodes[cur].arr[x])

       cur = nodes[cur].arr[x];

else

       cur = nodes[cur].arr[x] = free_head, free_head =
nodes[free_head].arr[52];

   if (free_head == free_tail)

       resize();
```

3. Memory Allocation

Instead of using a static or dynamic memory allocator, we used a combination of both. Static memory allocation decreases the run time but a lot of allocated space goes waste. Dynamic memory allocation increases the run time, even though it conserves space.

We implemented both and studied the trade-offs of using one over the other. However, we found the combination of both gives the best ‘score’.

We used the same trick used by the STL vector for memory allocation. Every time the memory falls short, we **double the size** of the data structure. This way, we only need to allocate the memory log(entries) number of times.

2 has been proven to be the most optimal factor as just O(N) space is allocated overall. With any other factor, this space increases.

```
bool kvStore::resize() {
int old_size = size;
    size <<= 1;
    TrieNode *new_nodes = (TrieNode *)calloc(size, sizeof(TrieNode));
    memcpy(new_nodes, nodes, sizeof(TrieNode) * old_size);
    delete[] nodes;
    nodes = new_nodes;
    nodes[free_tail].arr[52] = old_size;
    for (int i = old_size; i < size - 1; i++)
        nodes[i].arr[52] = i + 1;
    free_tail = size - 1;
    return true;
}
```

4. Memory usage

As we can see that the **“next” array will mostly have null pointers** if the puts are random enough. This was a huge **waste of memory**. To reduce this, we **used BSTs inside each of the trie nodes**. This way only the memory which is supposed to be used is used but at the cost of O(log(52)) = **O(8) search**. This way, the TPS reduces 8 times but the memory saved is a huge improvement in the overall score.

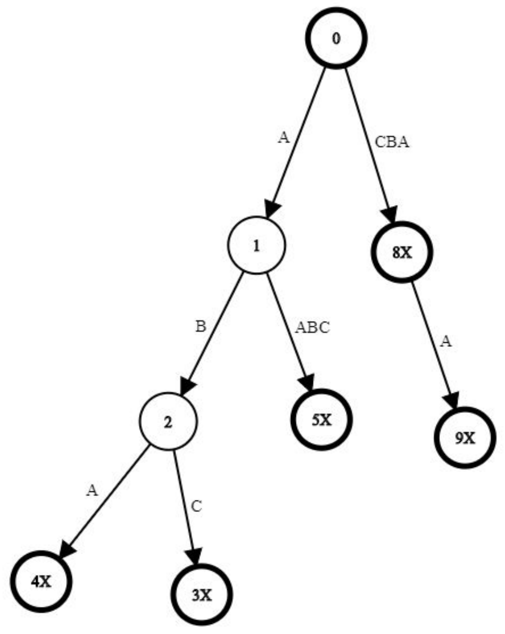
We experimented with **Red-Black Trees** and **AVL trees** and **unbalanced BSTs.** Most implementations online use the former two as their use cases rely on dictionary words. In our case, the keys are randomly generated and so the unbalanced BST implementation results in a **better runtime** because of a lack of rotations. The randomness also doesn’t let the tree height get too large on average and the tree remains balanced.

```
struct BSTNode {
   uint8_t left, right;
   char c;
   int data;
   int *ends;
};

class BST {
public:
      BSTNode nodes[52]; // max size of a BST is 52 - all characters
      int free_head, free_tail;
      int inorder(int cur,int &N); // BST functions
      bool insert(char c, int data,int * ends);
      int find(char c);
      bool remove(char c);
```

5. Compression

One further improvement we made was to use a **compressed trie** instead of a normal trie. We **stored more characters in a single edge** and use strcmp function to check if the string matches while querying, which is an optimized native C function. The trie above now looks like this:



As we have reduced the long single chains into strings, which **decreases the number of nodes we are using drastically**.

6. Slight optimizations:

6.1. Pragma

#pragma GCC optimize (string, ...)

This pragma allows you to set global optimization options for functions defined later in the source file.
We have used the “O2, Os, Ofast” flags as optimizations for speed and memory were most relevant for our use case.

We also experimented with flags such as “ffast-math” and “unroll-loops” but did not see a significant improvement.

```
#pragma GCC optimize("O2,Os,Ofast,no-signed-zeros,no-trapping-math”)
```

6.2 Register ints

Registers are faster than memory to access, so we put the variables which were most frequently used in our program in registers using *register* keyword. The keyword *register* hints to compiler that a given variable can be put in a register.

```
register int cur = 0;
register int lvl = 0;
```

6.3 Hot functions

Using gprof as well as a manual analysis of the code, we determined which functions were called the most during the execution phase. We then labeled them as hot functions, which tells the compiler to specially optimize them.

We labelled those functions as cold that are used infrequently.

```
BST()__attribute__((aligned(256),hot));
   bool resize()__attribute__((aligned(256),cold));
```

```
  int inorder(int &, int)__attribute__((aligned(256),hot));
  bool insert(uint8_t, int, int)__attribute__((aligned(256)));
  bool change_ends(uint8_t, int)__attribute__((aligned(256)));
  int find(uint8_t c)__attribute__((aligned(256),hot));
  int par(uint8_t c)__attribute__((aligned(256)));
  bool remove_bst(uint8_t c)__attribute__((aligned(256)));
```

6.4 Attribute aligning

Attribute aligning functions allows much faster access by registers. It also reduces the work done by the compiler.

```
 bool del(Slice &key, int, int, int)__attribute__((aligned(256)));
```

6.5 Uint8_t

uint8_t is the same as a byte. its shorthand for: a type of unsigned integer of length 8 bits. We used this in multiple places in our code where we didn't need to store an entire integer. This saved us a lot of memory.

```
uint8_t x;
encode(x, key.data[i]);
uint8_t node = nodes[cur].bst.find(x);
if (node == (uint8_t)-1)
     return false;
```

Our other implementations

We tried encoding the strings to bits and then saving the bit stream in the trie so that each node has **only 2 children**. This saves memory without needing any data structure inside each node for search operations.

However, we experimentally found out it takes a lot more time. We suspect this is because it increased the height of the trie by a factor of 6, thereby taking more time for get, put, and delete calls. Also, encoding every string and then decoding it takes extra cycles.

```
struct TrieNode {

   int arr[2];

   bool end = false;

   int ends = 0;

   int nxt;

   Slice *data;

};
```

A part of the get function highling how we use just a boolean at each node:

```
for (int j = 0; j < 6; j++, x >>= 1) {

      cur = nodes[cur].arr[x & 1];

      if (!cur)

                return false;

        }
```

References

1. https://15721.courses.cs.cmu.edu/spring2019/papers/08-oltpindexes2/p521-binna.pdf
2. http://www.cplusplus.com/reference/cstdlib/malloc/
3. https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html
4. https://docs.microsoft.com/en-us/cpp/preprocessor/optimize?view=vs-2019
5. https://www.cise.ufl.edu/~sahni/dsaaj/enrich/c16/tries.htm

Note: We did not follow any resources online to write the code. Everything has been written from scratch.