CS8.401 Software Programming for Performance

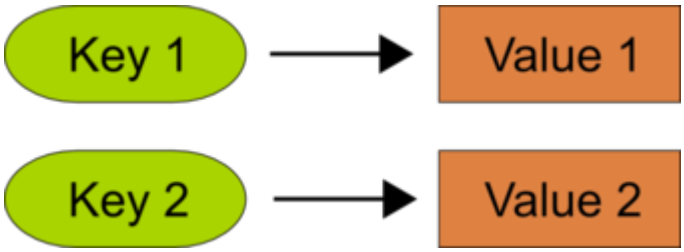
In-memory Key-Value Storage Software in C++

Team Members :

- Mallika Subramanian 2018101041
- Jyoti Sunkara 2018101044
- Ainsley D'Souza 2018101060
- Naren Akash R J 2018111020

Key-Value Store: An Introduction

A key-value store is a simple database that uses an associative array as the fundamental data model where each key is associated with one and only one value in a collection.



They provide a way to store, retrieve and update data using simple get, put and delete commands; the path to retrieve data is a direct request to the object in memory. The simplicity of this model makes a key-value store fast, easy to use, scalable, portable and flexible.

Analysis Of Data Structure

B Trees

B trees are not very efficient, when compared to other balanced trees, when they reside in RAM. Inserting or deleting elements involve moving many keys/values around. A B-tree has a chance to be efficient in RAM, if it has a very low branching factor, so that a node fits a cache line. This may lead to cache miss minimization.Leaf and non-leaf nodes are of different size, this complicates storage.

Hash table

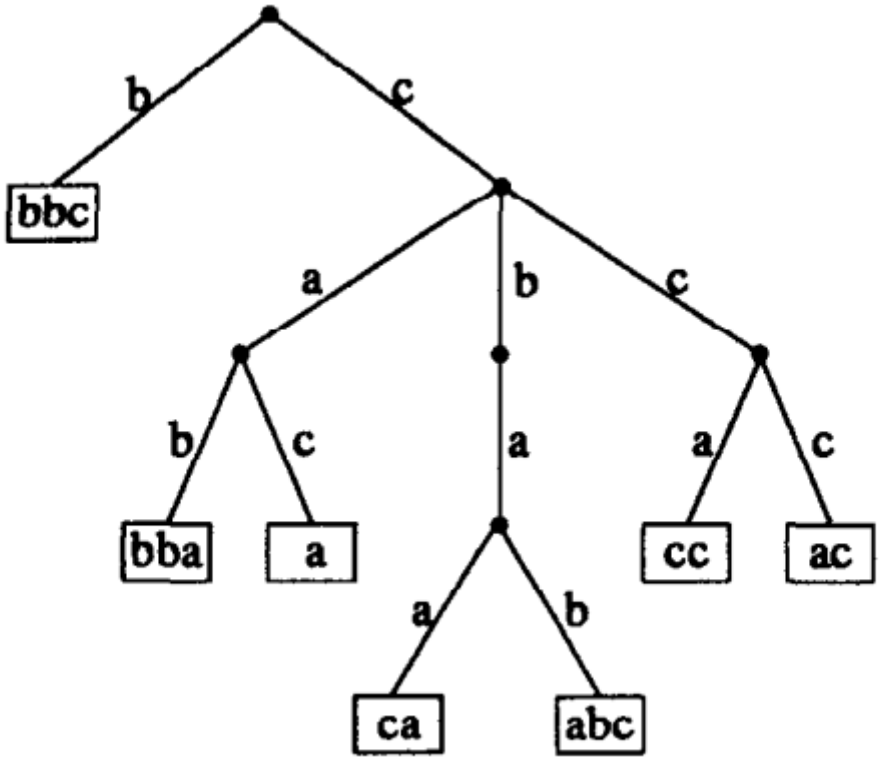
Hash tables become quite inefficient when there are many collisions. While extremely uneven hash distributions are extremely unlikely to arise by chance, a malicious adversary with knowledge of the hash function may be able to supply information to a hash that creates worst-case behavior by causing excessive collisions, resulting in very poor performance. In critical applications, universal hashing can be used; a data structure with better worst-case guarantees may be preferable.

Operations on a hash table take constant time on average.Thus hash tables are not effective when the number of entries is very small. It is slow due to synchronization.

Hash tables

Primary Data Structure: Trie

A trie is a multiway tree structure that stores sets of strings by successively partitioning them. Tries have two properties that cannot be easily imposed on data structures based on binary search.



First, strings are clustered by shared prefix. Second, there is a great reduction in the number of string comparisons—an important characteristic, as a major obstacle to efficiency is excessive numbers of string comparisons. Tries can be rapidly traversed and offer good worst-case performance, without the overhead of balancing.

Radix Trie

In computer science, a radix trie is a data structure that represents a space-optimized trie in which each node that is the only child is merged with its parent. This is done in order to save space.

In a trie, each node contains a character that is a part of the word. A radix trie tactfully breaks strings into common and uncommon strings and stores these substrings in the trie nodes.

In a large data set, after inserting the strings one can find common prefixes of length 4 to 10 characters in general, even if take the worst cade of 10 common character, the remainder of the string has a max length of 54 characters. Ordinarily in a regular trie we would require 54 nodes to store this. In a compressed trie this string can be accommodated in a single node thus saving a large amount of space.

The result is that the number of children of every internal node is at most the radix r of the radix trie, where r is a positive integer and a power x of 2, having $x \geq 1$.

In our key-value systems, the radix or base is the number of unique characters used to generate a key. Since a key can be made of both uppercase and lower case characters we can have a maximum of 26 + 26

children emerging from a parent node. However to ensure that each character is correctly mapped to its ASCII value we allocate a maximum space of 58 bytes for each trie node.

Unlike regular tries, edges can be labeled with sequences of elements as well as single elements. This makes radix tries much more efficient for large sets and for sets of strings that share long prefixes.

Common Operations in Compressed Trie

Compressed Trie supports the following main operations, all of which are `O(k)`, where k is the maximum length of all strings in the set:

Lookup:

Determines if a string is in the set. This operation is identical to tries except that some edges consume multiple characters.

Insert:

Add a string to the tree. We search the tree until we can make no further progress. At this point there are five possible cases,

> Case 1 Key to be inserted doesn't exist and had no common prefix in the database:

In this case the key is simply inserted via a pointer from the root node.

> Case 2

Key to be inserted already exits.

Here the trie traversed to reach the last node of the key and the value is overwritten. While traversing the trie for this key the descendants are incremented due to a condition in the search function.

To correct this appropriate node descendant values are decremented.

> Case 3

The key to be inserted is a super string of a node present in the database

Here, the common prefix is left untouched and the remaining part is attached is node to this common prefix.

> Case 4

The key to be inserted is a substring of a node in the database.

Here, the existing is broken into the common prefix and the remaining part which is attached as child to the common prefix node.

The descendants of the new and existing nodes are updated appropriately.

> Case 5

The key to be inserted has a common prefix with an existing node in the trie.

Here, the uncommon part of the existing node is broken into a new node and the key is inserted as a whole. The uncommon part of the key is then broken into a new node and then attached to the common prefix. Now the existing uncommon node is also attached to the common prefix node.

All splitting steps ensure that no node has more children than there are possible string characters.

Delete:

Delete a string from the trie. First, we traverse the trie until we find the key to be deleted. We then recursively backtrack the key path and remove the necessary nodes wherever possible. This is done in `O(k)` where `k` is the length of the key.

Lexicographical Storage

Thr array present in each node is of size `58` where the index corresponds to the alphabets in ASCII notation. This takes care of the sorting of keys.

API Overview

Supported API	Description
`get(key)`	Returns the value of the key
`put(key, value)`	Adds key-value, and overwrites the `existing value`
`delete(key)`	Delete the key from the data-structure
`get(int n)`	Returns the value of lexographically nth smaller key
`delete(int n)`	Delete the nth key-value pair

Performance Analysis

We ran our code with standard Linux performance analysis commands and obtained the following results:

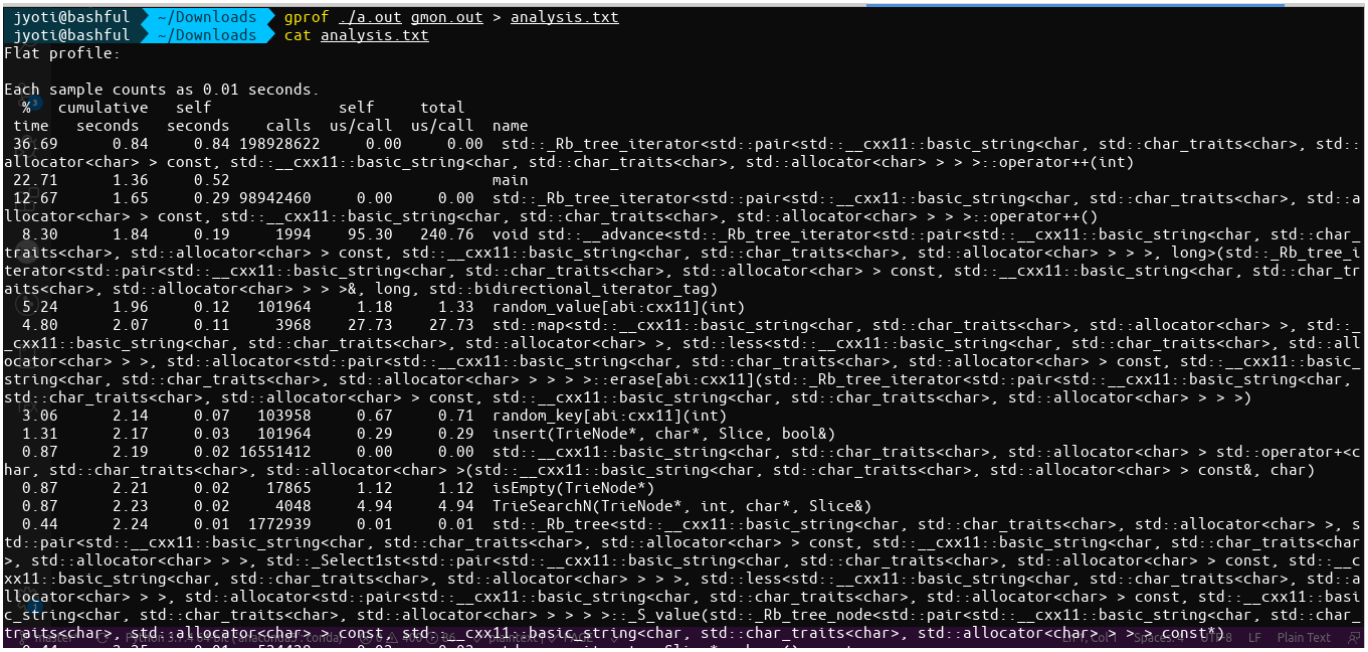
Valgrind

```
jyoti@bashful ~/Downloads valgrind --tool=cachegrind ./a.out
==23508== Cachegrind, a cache and branch-prediction profiler
==23508== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==23508== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==23508== Command: ./a.out
==23508==
--23508-- warning: L3 cache found, using its data for the LL simulation.
==23508== brk segment overflow in thread #1: can't grow to 0x4a37000
==23508== (see section Limitations in user manual)
==23508== NOTE: further instances of this message will not be shown
TIME FOR PUTTING 100000 ENTERIES = 2.3135 SEC
CORRECT OUTPUT
==23508==
==23508== I   refs:      24,933,908,616
==23508== I1  misses:           993,177
==23508== LLi misses:            90,995
==23508== I1  miss rate:           0.00%
==23508== LLi miss rate:           0.00%
==23508==
==23508== D   refs:      13,087,671,032  (8,926,177,686 rd   + 4,161,493,346 wr)
==23508== D1  misses:       383,504,642  (  379,440,430 rd   +     4,064,212 wr)
==23508== LLd misses:       144,951,926  (  141,780,001 rd   +     3,171,925 wr)
==23508== D1  miss rate:            2.9% (          4.3%     +           0.1%  )
==23508== LLd miss rate:            1.1% (          1.6%     +           0.1%  )
==23508==
==23508== LL refs:          384,497,819  (  380,433,607 rd   +     4,064,212 wr)
==23508== LL misses:        145,042,921  (  141,870,996 rd   +     3,171,925 wr)
==23508== LL miss rate:             0.4% (          0.4%     +           0.1%  )
```

Perf

```
jyoti@bashful ~/Downloads sudo perf stat ./a.out
TIME FOR PUTTING 100000 ENTERIES = 0.265988 SEC
CORRECT OUTPUT

 Performance counter stats for './a.out':

         34,728.39 msec task-clock                #    0.998 CPUs utilized
               615      context-switches          #    0.018 K/sec
                 7      cpu-migrations            #    0.000 K/sec
            63,230      page-faults               #    0.002 M/sec
    62,55,99,28,919      cycles                    #    1.801 GHz
    25,34,00,93,728      instructions              #    0.41  insn per cycle
     6,36,38,51,625      branches                  #  183.246 M/sec
       17,16,64,307      branch-misses             #    2.70% of all branches

      34.788339656 seconds time elapsed

      34.602746000 seconds user
       0.127995000 seconds sys
```

Gprof



Code Analysis

1. Benchmark Slice Struct

```
struct Slice
{
        uint8_t size;
        char *data;

        Slice()
        {
                data = (char *) malloc(64);
                strcpy(data,"\0");
                size = 0;
        }
};
```

`Slice` struct is used for storing the key and the value of every KV pair. The value of key or value along with its length are the components of the struct. By default, as defined in the constructor, the size is equal to 0 and the data value is a null string.

2. Compressed Trie Node Struct

```
struct TrieNode
{
        struct TrieNode* children[ALPHABET_SIZE];
        long long int descendants;
        bool isEndOfWord;
        char* value;
        Slice word;

        TrieNode() {
```

```
                value =(char *) malloc(64);
                word.data=(char *)malloc(256);
        }

        ~TrieNode() {
                cout<<value<<" destroyed"<<endl;
        }
};
```

Trie struct has:

- An array of Trie structs to store the children nodes,
- An integer variable for storing the number of end nodes at, or below that node,
- An boolean value which if true denotes that a Kv pair ends at that node,
- a string which stores the key ending at that node (if any) and
- a slice struct for storing the value associated with the key if the node is a terminal node.

3. Creating a new node

```
struct TrieNode* getNode(char* value)
{
        struct TrieNode* pNode = new TrieNode;
        strcpy(pNode->value,value);

        pNode->isEndOfWord = false;
        pNode->descendants = 0;

        for (int i = 0; i < ALPHABET_SIZE; i++)
                pNode->children[i] = NULL;

        return pNode;
}
```

When a new node is created in the radix trie, it has zero descendants and is defined to not be a end node. All the children node pointers are by default initialized to NULL.

4. Searching for a node

```
bool search(struct TrieNode* root, char* key,Slice &value, int add=0)
{
        struct TrieNode* pCrawl = root;

        while(pCrawl != NULL && *key != '\0')
        {
                int index = (*key) - 'A';

                if (!pCrawl->children[index])
                {
```

```
                        return false;
                }
                else
                {
                        pCrawl->descendants+=add;

                        if(strcmp(key,pCrawl->children[index]->value) == 0)
                        {
                                strcpy(value.data,pCrawl->children[index]-
>word.data);
                                value.size=pCrawl->children[index]-
>word.size;

                                l->children[index]->isEndOfWord<<endl;
                                return (pCrawl->children[index]-
>isEndOfWord);
                        }
                        else
                        {
                                int size = 0;

                                char *temp1 = key, *temp2 = pCrawl-
>children[index]->value;
                                while(*temp1 == *temp2 && *temp1 != '\0' &&
*temp2 != '\0')
                                {
                                        size++;
                                        temp1++;
                                        temp2++;
                                }
                                key += size;
                                pCrawl = pCrawl->children[index];
                        }
                }
        }
}
```

Starting from the root node, recursively go down each layer and look for a key match with the key words terminating at that node under consideration. If matched and the node is a terminal node, we output True.

5. Inserting a new node

```
void insert(struct TrieNode* root, Slice key, Slice value, bool &f)
{
        struct TrieNode* pCrawl = root;
        char rand[] = "";

        if(*(key.data) == '\0')
                return;
        else
        {
```

```
                int index = *(key.data) - 'A';
                pCrawl->descendants += 1;

                if (!pCrawl->children[index])
                {
                        char str[key.size+1];
                        strcpy(str,key.data);
                        str[key.size] = '\0';

                        pCrawl->children[index] = getNode(str);
                        strcpy(pCrawl->children[index]->value, str);

                        pCrawl->children[index]->isEndOfWord = true;
                        strcpy(pCrawl->children[index]-
>word.data,value.data);
                        pCrawl->children[index]->word.size=value.size;
                }
                else
                {
                        int size = 0;
                        char *temp1 = key.data, *temp2 = pCrawl-
>children[index]->value;

                        while(*temp1 == *temp2 && *temp1 != '\0' && *temp2
!= '\0')
                        {
                                size++;
                                temp1++;        temp2++;
                        }

                        char str1[64], str2[64];
                        strcpy(str1,temp1);
                        strcpy(str2,temp2);

                        if(*temp1 != '\0' && *temp2 != '\0')
                        {
                                strcpy(pCrawl->children[index]->value,
str2);
                                pCrawl->descendants -= 1;
                                void * temp = pCrawl->children[index];
                                pCrawl->children[index] = NULL;

                                insert(pCrawl, key, value,f);

                                Slice tempslice;

                                strcpy(tempslice.data,str1);
                                tempslice.size=(uint8_t)strlen(str1);
                                insert(pCrawl->children[index], tempslice,
value,f);
                                pCrawl->children[index]->isEndOfWord =
false;

                                pCrawl->children[index]->children[str2[0]-
```

```
'A'] = (TrieNode*)temp;
                                pCrawl->children[index]->descendants +=
pCrawl->children[index]->children[str2[0]-'A']->descendants;
                                if(pCrawl->children[index]-
>children[str2[0]-'A']->isEndOfWord)
                                        pCrawl->children[index]-
>descendants += 1;

                        }
                        else if(*temp1 == '\0' && *temp2 != '\0')
                        {
                                pCrawl->children[index]->isEndOfWord =
true;
                                strcpy(pCrawl->children[index]->value,
str2);
                                pCrawl->descendants -= 1;
                                void * temp = pCrawl->children[index];
                                pCrawl->children[index] = NULL;

                                insert(pCrawl, key,value,f);

                                pCrawl->children[index]->children[str2[0]-
'A'] = (TrieNode*)temp;
                                pCrawl->children[index]->descendants +=
pCrawl->children[index]->children[str2[0]-'A']->descendants ;
                                if(pCrawl->children[index]-
>children[str2[0]-'A']->isEndOfWord)
                                        pCrawl->children[index]-
>descendants += 1;
                        }
                        else if(*temp2 == '\0' && *temp1 != '\0')
                        {
                                Slice tempslice;

                                strcpy(tempslice.data,str1);
                                tempslice.size=(uint8_t)strlen(str1);
                                insert(pCrawl->children[index],
tempslice,value,f);
                        }
                        else
                        {
                                pCrawl->children[index]->word = value;
                                if(pCrawl->children[index]->isEndOfWord ==
true)
                                        f=true;
                                pCrawl->children[index]->isEndOfWord =
true;
                        }


                        char str[size+1];
                        strncpy(str,pCrawl->children[index]->value,size);
                        str[size] = '\0';
                        strcpy(pCrawl->children[index]->value,str);
```

```
                }
        }
}
```

6. Remove a node from the trie

```
TrieNode* TrieRemove(TrieNode* root, Slice key, uint8_t depth = 0)
{
        // If tree is empty
        if (!root)
                return NULL;

        // If last character of key is being processed
        if (depth == key.size)
        {
                // This node is no more end of word after
                // removal of given key
                if (root->isEndOfWord)
                        root->isEndOfWord = false;

                // If given is not prefix of any other word
                if (isEmpty(root)) {
                        delete (root);
                        root = NULL;
                }

                return root;
        }

        // If not last character, recur for the child
        // obtained using ASCII value
        int index = key.data[depth] - 'A';

        int size = 0;
        char *temp1 = key.data + depth, *temp2 = root->children[index]-
>value;
        while(*temp1 == *temp2 && *temp1 != '\0' && *temp2 != '\0')
        {
                size++;
                temp1++;        temp2++;
        }
        if(*temp2 == '\0')
                root->children[index] = TrieRemove(root->children[index],
key, depth + size);

        // If root does not have any child (its only child got
        // deleted), and it is not end of another word.
        if (isEmpty(root) && root->isEndOfWord == false)
        {
                delete (root);
                root = NULL;
```

```
        }
        else
                root->descendants -= 1;

        return root;
}
```

Search through the nodes and find out the node where the desired key terminates. If the parent of that node had only one other child, remove that parent node too and add the other node to the grand-parent node of the respective terminal nodes.

7. Searching the lexicographically Nth smallest key

```
void TrieSearchN(struct TrieNode *root, int n, char * str, Slice value)
{
        int x = 0, count = 0;
        int *current = &x;

        struct TrieNode *pCrawl = root;
        struct TrieNode *temp = NULL;

        while(*current < n && pCrawl != NULL)
        {
                temp = NULL;

                for (int i = 0; i < ALPHABET_SIZE && temp == NULL ; i++)
                {
                        if (pCrawl->children[i])
                        {
                                if(pCrawl->children[i]->isEndOfWord ==
true)
                        {
                                *current += 1;
                        }


                        int sum = *current + pCrawl->children[i]-
>descendants;

                        if(sum < n)
                                *current = sum;

                        else if(sum >= n)
                        {
                                temp = pCrawl->children[i];
                        }
                        }
                }
                pCrawl = temp;
                if(pCrawl != NULL)
                        strcat(str,pCrawl->value);
```

```
                count++;
        }
        strcpy(value.data, pCrawl->word.data);
        value.size = pCrawl->word.size;
        value = pCrawl->word;
}
```

The SearchN function starts from the root node and for each of the children nodes, in ascending order of ASCII values, it calculates the total number of descendants till then to find the lexicographic Nth smallest key.

8. Removing the lexograpichally Nth smallest key

```
void TrieRemoveN(TrieNode* root, int N)
{
        char str[64] = "\0";
        Slice value;

        TrieSearchN(root,N,str,value);

        Slice key;

        strcpy(key.data,str);
        key.size=(uint8_t)strlen(str);
        TrieRemove(root,key);
}
```

The SearchN function is used to find the key of the lexicographically Nth smallest key and the KV pair is removed from the radix trie using the TrieRemove function with the obtained key as a parameter.

Trie Hybrids

Layered trie

Searching a trie T for a key w just requires tracing a path down the trie as follows: at depth i, the ith digit of w is used to orientate the branching. We need to specify which search structure is used to choose the correct sub-trie within a node.

1. Array-Trie Hybrid

Uses an array of pointers to sub-tries. In case of a large alphabet too many empty pointers are created.

2. Linked List-Trie Hybrid

Linked list of sub-tries reduces the high storage requirement of the array-trie hybrid but traversal operations are expensive.

3. AVL-Trie Hybrid

AVL Trees have the time efficiency of arrays and the space efficiency of the linked lists.

References

1. The analysis of hybrid trie structures, Clément (Julien), Flajolet (Philippe), and Vallée (Brigitte).
2. Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache, Nikolas Askitis, and Justin Zobel
3. Emory CS323 Data Structures and Algorithms: Compressed(Patricia) Tries