SPP Final Project: Implementation Specifications

Team: Optimiser Prime

Members:
1. Pranav Tadimeti (2018101055)
2. Nishant Sachdeva (2018111040)
3. Pranav Kirsur(2018101070)
4. Aman Kumar Kashyap(2019121010)

Requirements:

1. Maximum size of the keys will be 64 bytes (52 possible characters: a-z and A-Z).
2. Maximum size of the values will be 256 bytes.
3. 10 million put* requests (at the time of initialisation) expected to be completed within 2 minutes(maximum).

(* put : The "put" request is a member function of the skeleton class given to us in the initial project specifications.)

Previous Specs (for reference) :

1. We were looking to employ cache memory optimisations for the storage with some encoding. Below are the reasons why :
   a. The 48 bit encoding is very good for cache locality and cache hit ratio is improved by a factor of 4/3.
   b. Having the 48 bit encoding allows us to (With the help of some Pragma Directives to help with aligning) use SIMD instructions for executing compare instructions quickly. Here's how:
      i. We keep a separate list of all the keys so that we can speed up the search queries.
      ii. For most modern day processors, cache lines are 64 bytes each. At the outset of the code, we use Pragma directives to ask for our list to be aligned to a 64 byte addresses.

iii. How this helps is that : Since our words are 48 bits in size, and we aligned them to a 64 bit starting address, we will have distributions of the form :

(48 + 16) : (32 + 32) : (16 + 48)

Now that we know the alignment of the words in the cache file, we can use SIMD instructions to do quick comparisons between the words. (Since the starting address was on a 64 bit aligned cache line and our word size if 48 bits, we understand that all our words will be starting on 16 bit aligned cache addresses) .

The process of comparisons goes as follows:

1. SIMD allows us to do 16 bit compares and 32 bit compares.
2. Since our word size is 48 and every address is 16 bit aligned, we will execute two comparisons, first of which will be a 16 bit comparison : to be followed by a 32 bit comparison on the remaining bits.
3. This trick will allow us to finish comparing two words in just 2 instructions. This is a massive speed up because normally it would take us 48 instructions to actually compare 2 words (both of whom are 48 bits long).

2. There were many algorithms considered for the actual implementation of the final storage mechanism. Below are some the structures that were considered :

1. B-Tree It is a balanced search tree. Each node of B-Tree stores multiple key-values. B-Tree of order m has at max m-1 keys in one node. Most operations like insert, delete, update take O(height) time. The height of the tree decreases exponentially with respect to the number of key value pairs, that is, height = logm n where n is the number of key value pairs. So if n = 10^5 and m =10, height will be 5. Thus, most operations will be very fast. For each node, we need to do a binary search to find the right node we are looking for. Therefore at each level, we take log2m time to search in a given node using binary search. Therefore the total time taken is O(h * log2m). We shall find the optimal value of m to give the best runtime.
2. . Balanced BST It is an ordered data structure. The operations take O(height) time. We will consider this as an alternative approach to maintain ordering of key value pairs.

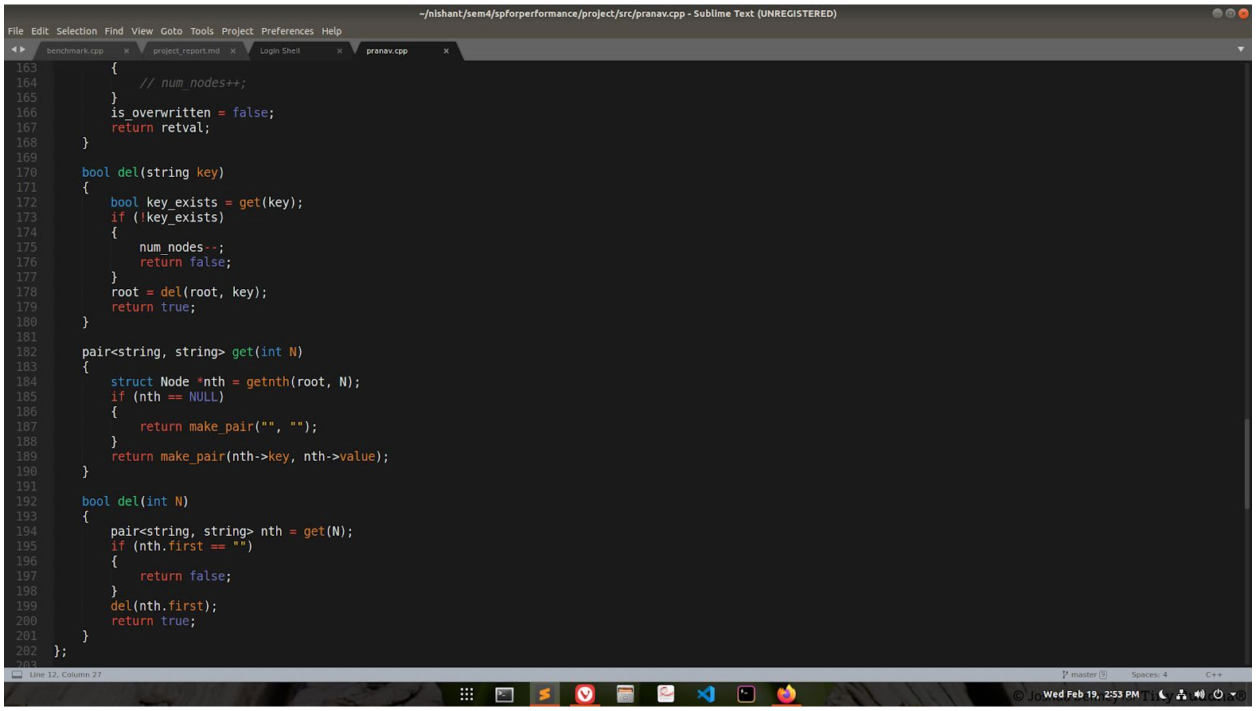
3. Hash Table + Tree: Hash table with a suitable hashing function will be used for fast lookup and insertion. Ordering of the keys will be maintained by a tree. The advantage of this method is that the normal operations will require a key as parameter, so by hashing the key and getting the value from the hash table, we can perform the operation in O(1) amortized. For the operations dependent on lexicographical order, we shall use a tree based data structure for finding the k th key. Thus most operations will be very fast except for the ones that require getting the key ordering. It is expected that this approach will perform the best.
4. Tries Since we want to index by key, it makes sense to use a data structure such as a compressed trie. We store the value of at the leaves of the trie. We also maintain a count of the leaves in the nodes to index by number. The advantage of this method is that it will consume less memory as we get a lot of compression.
5. FST We are also looking at FSTs. It seems appropriate for this purpose. It seems implementation heavy hence we may or may not use it.

All of these cases were considered thoroughly and a certain mix of the algorithm and cache optimisation approach was adapted. Below are the details of the final implementation :

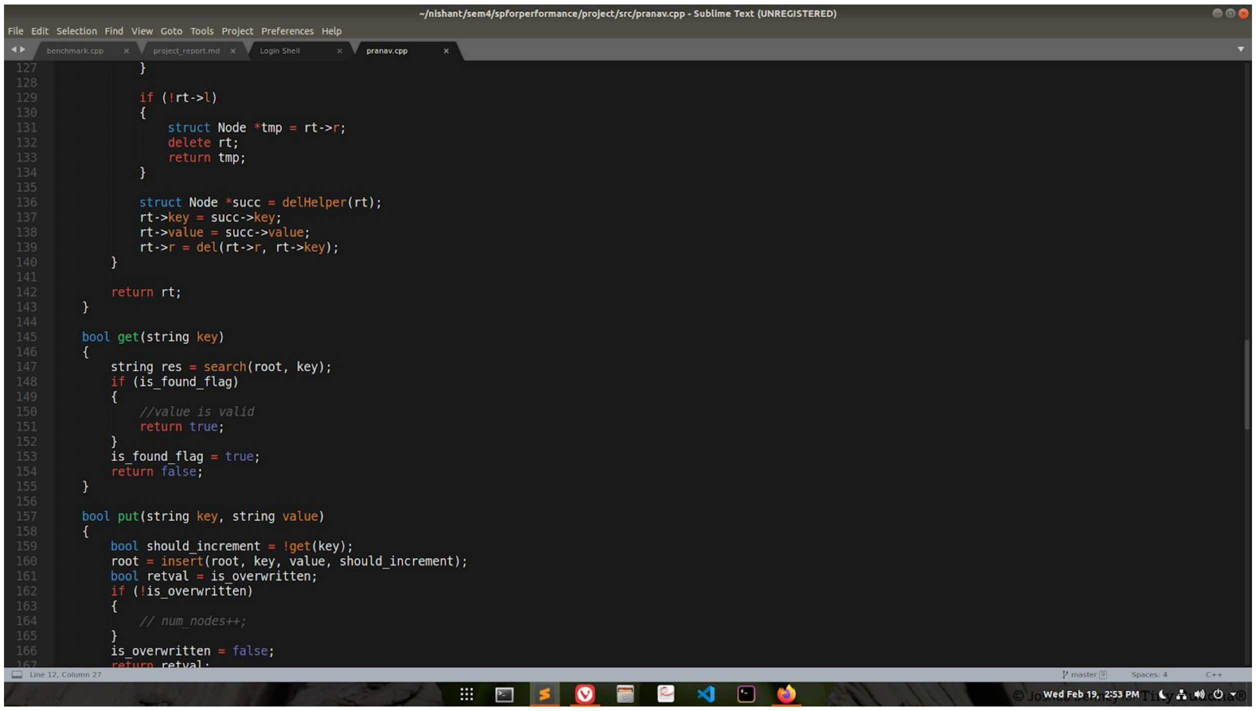
Details of the Final Implementation :

1. The final code has been implemented using an Augmented BST. The major reason behind taking that decision is the speed which the BST brings to the table (this speed is the primary reason why the extra memory that gets used with the BST was treated as an unavoidable collateral damage)
2. The reason behind using an Augmented BST is that we are going get queries that are of the format where we will be asked for a certain Nth key. In those cases, we will need to find that element based on the order as decided by the strcmp function.

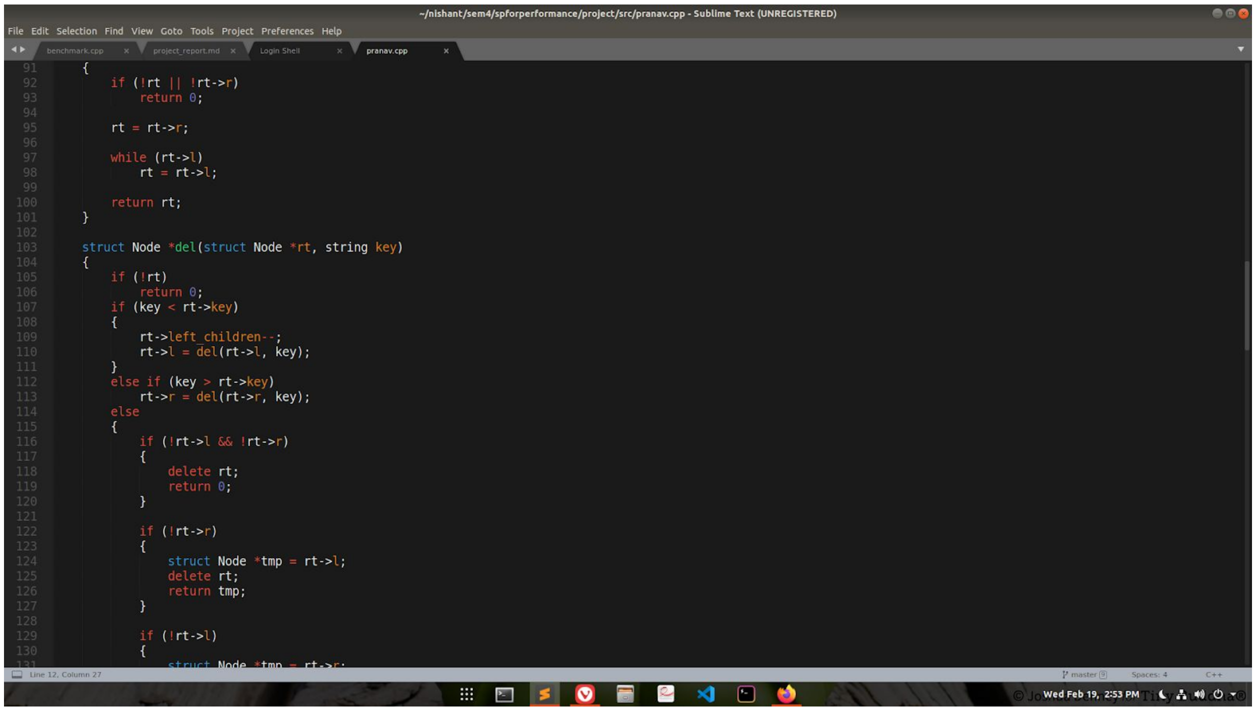
Below are some code snippets with explanations (if deemed necessary) :

1. 
~/nishant/sem4/spforperformance/project/src/pranav.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
benchmark.cpp
project_report.md
Login Shell
pranav.cpp
Line 12, Column 27
master
Spaces: 4
C++
Wed Feb 19, 2:53 PM

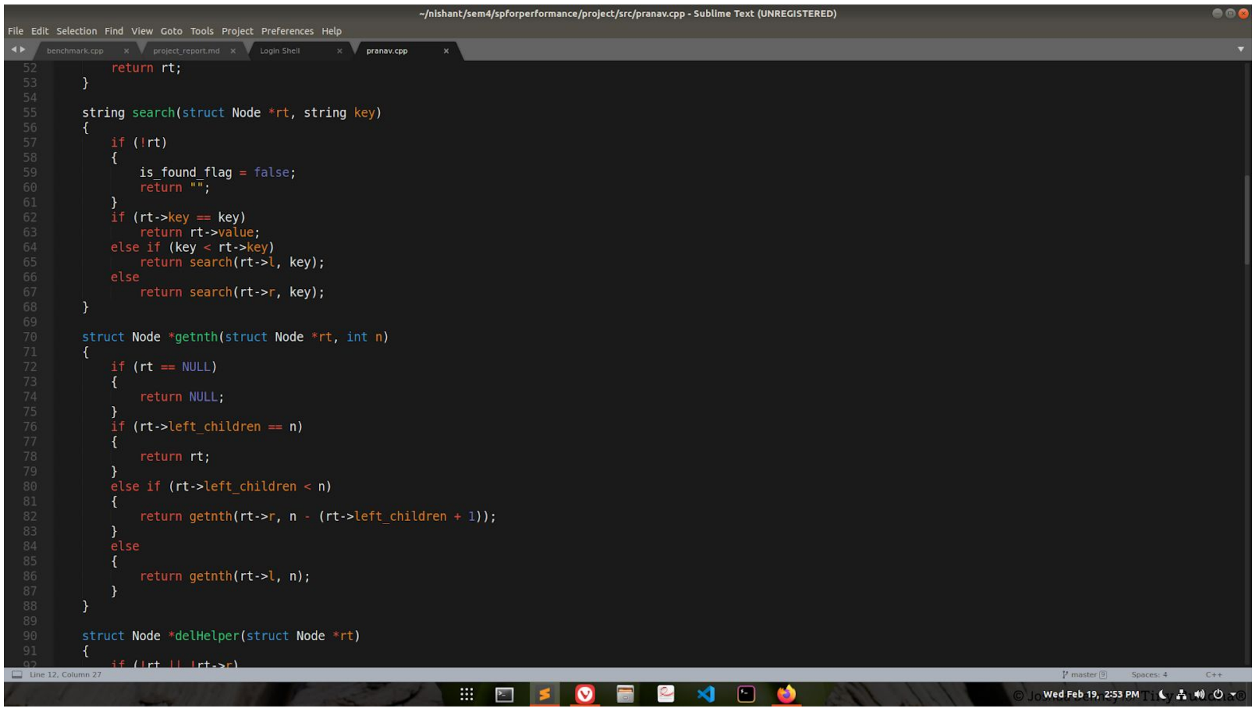
```cpp
        {
            // num_nodes++;
        }
        is_overwritten = false;
        return retval;
    }

    bool del(string key)
    {
        bool key_exists = get(key);
        if (!key_exists)
        {
            num_nodes--;
            return false;
        }
        root = del(root, key);
        return true;
    }

    pair<string, string> get(int N)
    {
        struct Node *nth = getnth(root, N);
        if (nth == NULL)
        {
            return make_pair("", "");
        }
        return make_pair(nth->key, nth->value);
    }

    bool del(int N)
    {
        pair<string, string> nth = get(N);
        if (nth.first == "")
        {
            return false;
        }
        del(nth.first);
        return true;
    }
};
```


~/nishant/sem4/spforperformance/project/src/pranav.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
benchmark.cpp
project_report.md
Login Shell
pranav.cpp
Line 12, Column 27
master
Spaces: 4
C++
Wed Feb 19, 2:53 PM

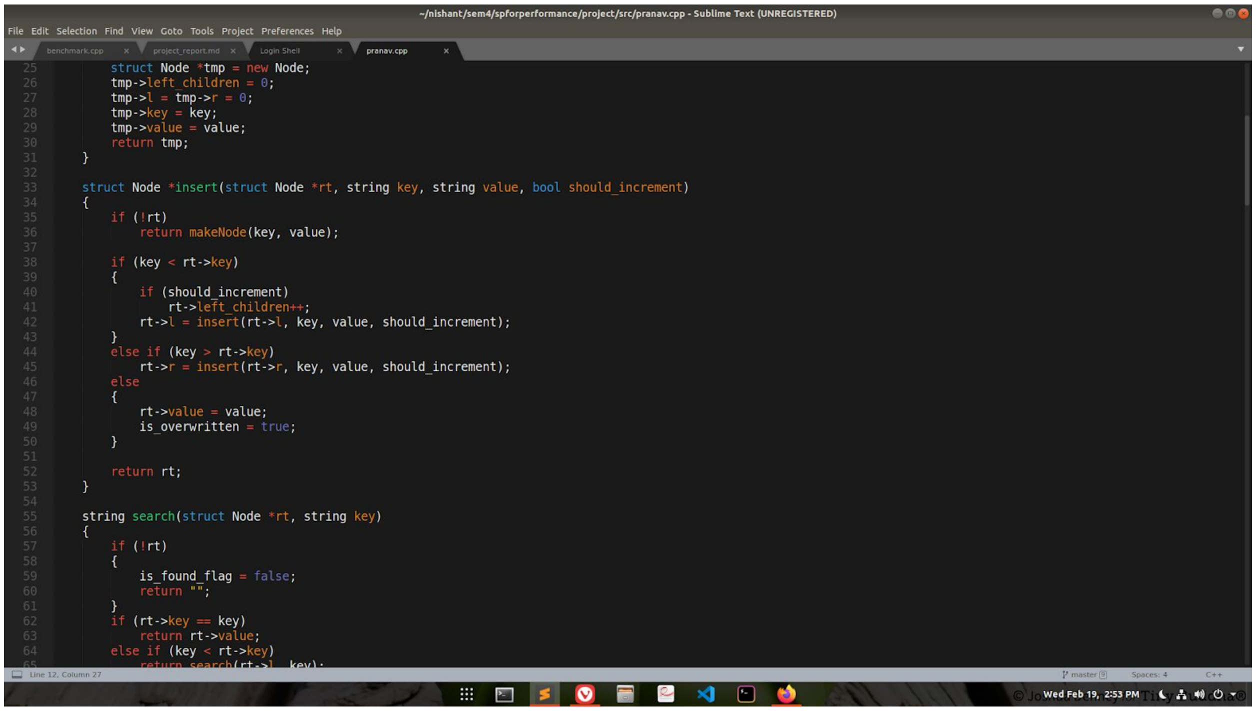
```cpp
            }

            if (!rt->l)
            {
                struct Node *tmp = rt->r;
                delete rt;
                return tmp;
            }

            struct Node *succ = delHelper(rt);
            rt->key = succ->key;
            rt->value = succ->value;
            rt->r = del(rt->r, rt->key);
        }

        return rt;
    }

    bool get(string key)
    {
        string res = search(root, key);
        if (is_found_flag)
        {
            //value is valid
            return true;
        }
        is_found_flag = true;
        return false;
    }

    bool put(string key, string value)
    {
        bool should_increment = !get(key);
        root = insert(root, key, value, should_increment);
        bool retval = is_overwritten;
        if (!is_overwritten)
        {
            // num_nodes++;
        }
        is_overwritten = false;
        return retval;
```


~/nishant/sem4/spforperformance/project/src/pranav.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
benchmark.cpp
project_report.md
Login Shell
pranav.cpp
Line 12, Column 27
master
Spaces: 4
C++
Wed Feb 19, 2:53 PM

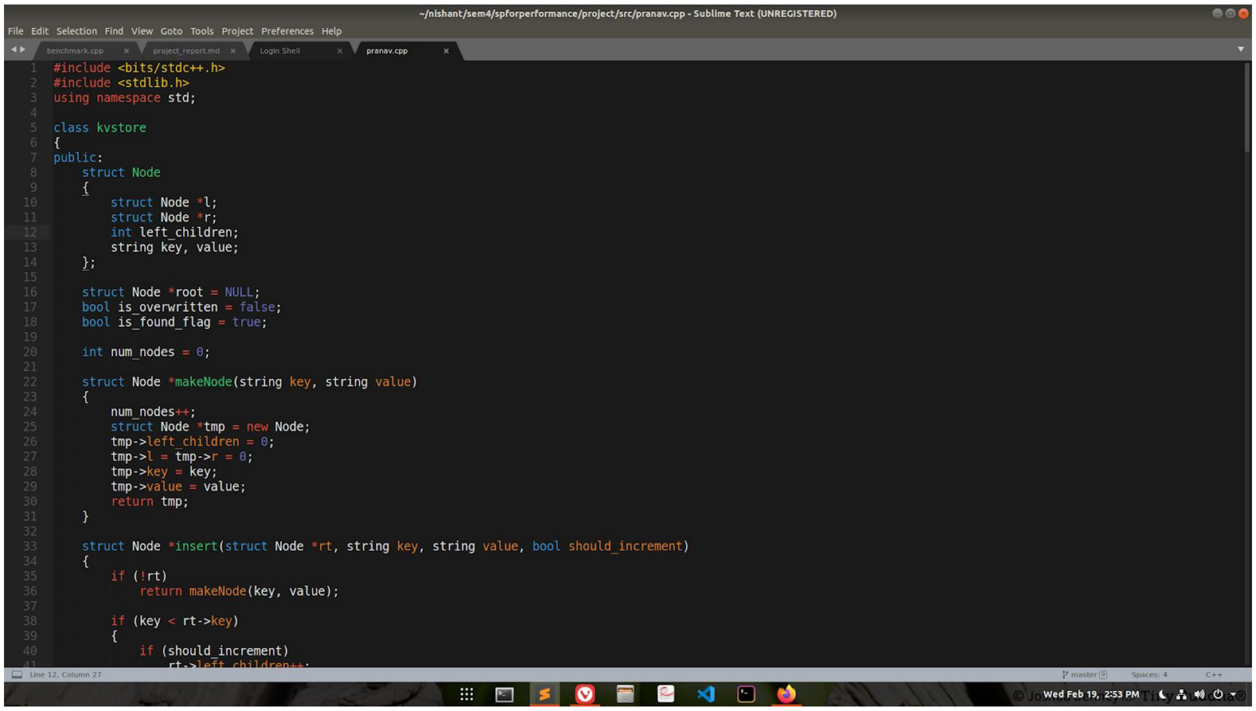
```
    {
        if (!rt || !rt->r)
            return 0;

        rt = rt->r;

        while (rt->l)
            rt = rt->l;

        return rt;
    }

    struct Node *del(struct Node *rt, string key)
    {
        if (!rt)
            return 0;
        if (key < rt->key)
        {
            rt->left_children--;
            rt->l = del(rt->l, key);
        }
        else if (key > rt->key)
            rt->r = del(rt->r, key);
        else
        {
            if (!rt->l && !rt->r)
            {
                delete rt;
                return 0;
            }

            if (!rt->r)
            {
                struct Node *tmp = rt->l;
                delete rt;
                return tmp;
            }

            if (!rt->l)
            {
                struct Node *tmp = rt->r;
```


~/nishant/sem4/spforperformance/project/src/pranav.cpp - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
benchmark.cpp
project_report.md
Login Shell
pranav.cpp
Line 12, Column 27
master
Spaces: 4
C++
Wed Feb 19, 2:53 PM

```
        return rt;
    }

    string search(struct Node *rt, string key)
    {
        if (!rt)
        {
            is_found_flag = false;
            return "";
        }
        if (rt->key == key)
            return rt->value;
        else if (key < rt->key)
            return search(rt->l, key);
        else
            return search(rt->r, key);
    }

    struct Node *getnth(struct Node *rt, int n)
    {
        if (rt == NULL)
        {
            return NULL;
        }
        if (rt->left_children == n)
        {
            return rt;
        }
        else if (rt->left_children < n)
        {
            return getnth(rt->r, n - (rt->left_children + 1));
        }
        else
        {
            return getnth(rt->l, n);
        }
    }

    struct Node *delHelper(struct Node *rt)
    {
        if (!rt || !rt->r)
```



```cpp
        struct Node *tmp = new Node;
        tmp->left_children = 0;
        tmp->l = tmp->r = 0;
        tmp->key = key;
        tmp->value = value;
        return tmp;
    }

    struct Node *insert(struct Node *rt, string key, string value, bool should_increment)
    {
        if (!rt)
            return makeNode(key, value);

        if (key < rt->key)
        {
            if (should_increment)
                rt->left_children++;
            rt->l = insert(rt->l, key, value, should_increment);
        }
        else if (key > rt->key)
            rt->r = insert(rt->r, key, value, should_increment);
        else
        {
            rt->value = value;
            is_overwritten = true;
        }

        return rt;
    }

    string search(struct Node *rt, string key)
    {
        if (!rt)
        {
            is_found_flag = false;
            return "";
        }
        if (rt->key == key)
            return rt->value;
        else if (key < rt->key)
            return search(rt->l, key);
```



```cpp
#include <bits/stdc++.h>
#include <stdlib.h>
using namespace std;

class kvstore
{
public:
    struct Node
    {
        struct Node *l;
        struct Node *r;
        int left_children;
        string key, value;
    };

    struct Node *root = NULL;
    bool is_overwritten = false;
    bool is_found_flag = true;

    int num_nodes = 0;

    struct Node *makeNode(string key, string value)
    {
        num_nodes++;
        struct Node *tmp = new Node;
        tmp->left_children = 0;
        tmp->l = tmp->r = 0;
        tmp->key = key;
        tmp->value = value;
        return tmp;
    }

    struct Node *insert(struct Node *rt, string key, string value, bool should_increment)
    {
        if (!rt)
            return makeNode(key, value);

        if (key < rt->key)
        {
            if (should_increment)
                rt->left_children++;
```

As we have seen, this is the implementation of the code for the storage and retrieval of the the key-value storage pairs in C++. Changes will be made to the algo of implementation and to the code itself as deemed fit.