Sorted Student Map Project Design/Implementation

By: Janiyah Cutchin-Brown

Project Overview

This project is a design which uses a map structure to store student names and IDs. It maintains students sorted alphabetically by name for fast searching. The operations it supports are: Add, Get, Remove, and Count students by name prefix.

Motivation

This algorithm is great for managing student records in a sorted structure enables. It makes for faster lookups using binary search and efficient range queries. It is also useful for real-world cases such as enrollment systems, school databases, and directory services.

Design

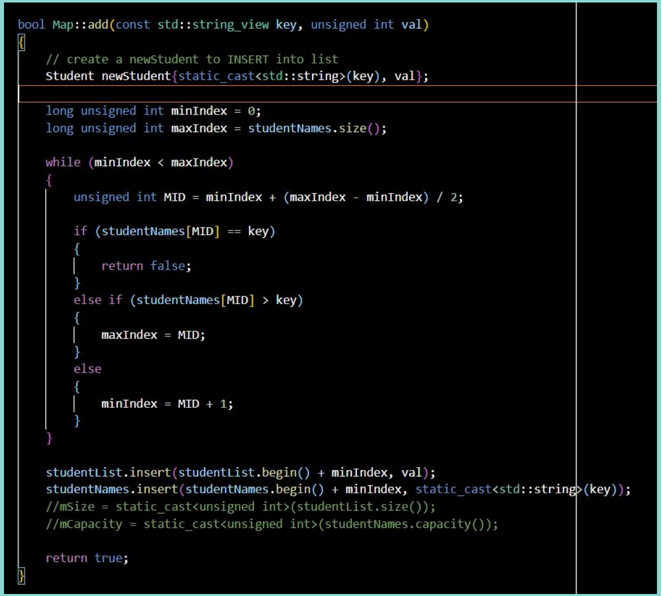
The program utilizes two parallel vectors.

1. studentNames (std::vectorstd::string) - stores sorted student names
2. studentList (std::vector<unsigned int>)- stores corresponding student IDs

A sorted insertion is what keeps the student names in alphabetical order. A binary search is used for lookups, enabling O(log n) complexity

Key Operations—Add

This operation uses a binary search to find the correct insertion point. It rejects duplicates if student name already exists. It inserts student name and ID at the sorted position and maintains that order after each insertion.

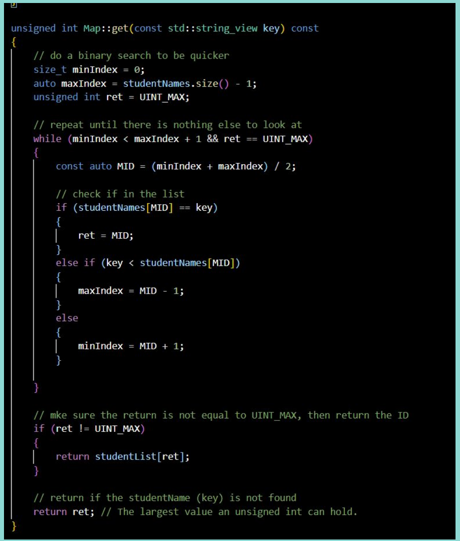


```cpp
bool Map::add(const std::string_view key, unsigned int val)
{
    // create a newStudent to INSERT into list
    Student newStudent{static_cast<std::string>(key), val};

    long unsigned int minIndex = 0;
    long unsigned int maxIndex = studentNames.size();

    while (minIndex < maxIndex)
    {
        unsigned int MID = minIndex + (maxIndex - minIndex) / 2;

        if (studentNames[MID] == key)
        {
            return false;
        }
        else if (studentNames[MID] > key)
        {
            maxIndex = MID;
        }
        else
        {
            minIndex = MID + 1;
        }
    }

    studentList.insert(studentList.begin() + minIndex, val);
    studentNames.insert(studentNames.begin() + minIndex, static_cast<std::string>(key));
    //mSize = static_cast<unsigned int>(studentList.size());
    //mCapacity = static_cast<unsigned int>(studentNames.capacity());

    return true;
}
```

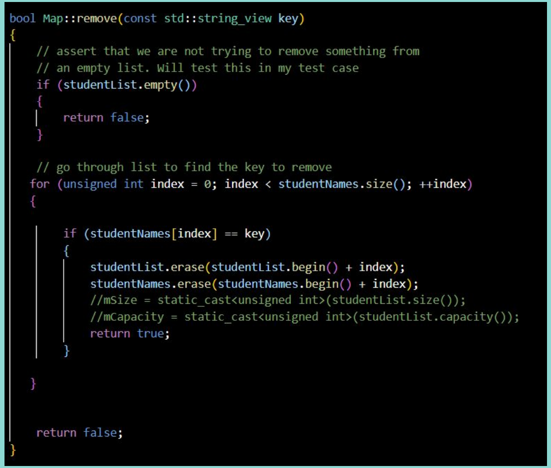
Key Operations—Get & Remove

The Get function performs a binary search on studentNames to find the index and return the corresponding ID.

The Remove function performs a linear search to find and remove a student by name from both vectors.



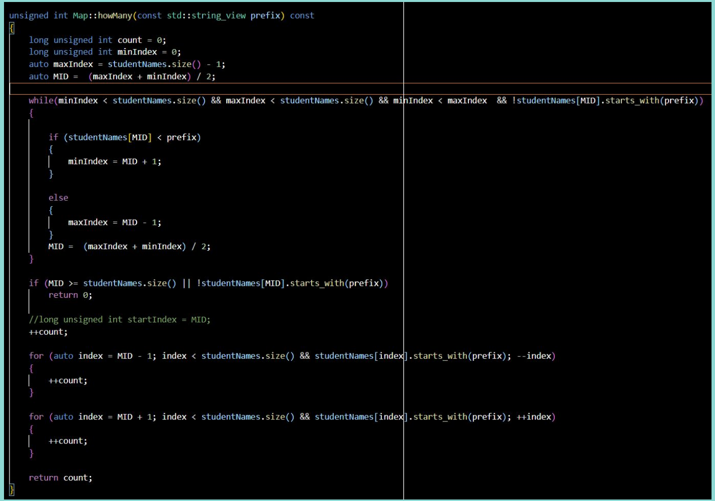
```cpp
unsigned int Map::get(const std::string_view key) const
{
    // do a binary search to be quicker
    size_t minIndex = 0;
    auto maxIndex = studentNames.size() - 1;
    unsigned int ret = UINT_MAX;

    // repeat until there is nothing else to look at
    while (minIndex < maxIndex + 1 && ret == UINT_MAX)
    {
        const auto MID = (minIndex + maxIndex) / 2;

        // check if in the list
        if (studentNames[MID] == key)
        {
            ret = MID;
        }
        else if (key < studentNames[MID])
        {
            maxIndex = MID - 1;
        }
        else
        {
            minIndex = MID + 1;
        }

    }

    // mke sure the return is not equal to UINT_MAX, then return the ID
    if (ret != UINT_MAX)
    {
        return studentList[ret];
    }

    // return if the studentName (key) is not found
    return ret; // The largest value an unsigned int can hold.
}
```



```cpp
bool Map::remove(const std::string_view key)
{
    // assert that we are not trying to remove something from
    // an empty list. Will test this in my test case
    if (studentList.empty())
    {
        return false;
    }

    // go through list to find the key to remove
   for (unsigned int index = 0; index < studentNames.size(); ++index)
   {

        if (studentNames[index] == key)
        {
            studentList.erase(studentList.begin() + index);
            studentNames.erase(studentNames.begin() + index);
            //mSize = static_cast<unsigned int>(studentList.size());
            //mCapacity = static_cast<unsigned int>(studentList.capacity());
            return true;
        }

   }


    return false;
}
```

Additional Function–howMany(prefix)

The howMany function returns the count of students whose names with a prefix given by the user. It uses a binary search to locate the first occurrence of the prefix and counts all matching names forward and backward from that position.



```cpp
unsigned int Map::howMany(const std::string_view prefix) const
{
    long unsigned int count = 0;
    long unsigned int minIndex = 0;
    auto maxIndex = studentNames.size() - 1;
    auto MID =  (maxIndex + minIndex) / 2;

    while(minIndex < studentNames.size() && maxIndex < studentNames.size() && minIndex < maxIndex  && !studentNames[MID].starts_with(prefix))
    {

        if (studentNames[MID] < prefix)
        {
            minIndex = MID + 1;
        }

        else
        {
            maxIndex = MID - 1;
        }
        MID =  (maxIndex + minIndex) / 2;
    }

    if (MID >= studentNames.size() || !studentNames[MID].starts_with(prefix))
        return 0;

    //long unsigned int startIndex = MID;
    ++count;

    for (auto index = MID - 1; index < studentNames.size() && studentNames[index].starts_with(prefix); --index)
    {
        ++count;
    }

    for (auto index = MID + 1; index < studentNames.size() && studentNames[index].starts_with(prefix); ++index)
    {
        ++count;
    }

    return count;
}
```

Challenges and Solutions

Some challenges and solutions in implementing this algorithm include maintaining sorted order efficiency with vector insertions, handling duplicates gracefully, designing the prefix count with careful boundary checks and the use of std::string_view for efficient string referencing.

Results and Testing

- Basic tests confirm add/get/remove correctness
- Successfully handles thousands of student records
- Demonstrates O(log n) search times with binary search

Lessons Learned

- The importance of balancing data structure choice and operation complexity
- Benefits of separating keys and values into parallel arrays for sorting
- Leveraging standard library functions like std::shuffle and std::string_view
- Challenges of careful indexing and boundary handling in binary search

Next Steps

Some next steps in utilizing this algorithm include optimising the remove operation to use a binary search instead of linear, extend to support updates to student IDs, add more robust testing, and explore alternative data structures such as hash maps or balanced trees