1. **Цель лабораторной работы**

Целью данной работы является определение основных критериев оценки структуры данных и методов ее обработки применительно к конкретной задаче.

1. **Описание задания**
   1. **Задание**

Дано N элементов, где каждый элемент является предложением на естественном языке.

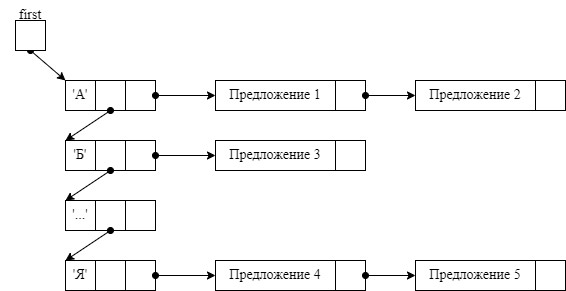
* 1. **Основные требования**

Основной вариант задания включает в себя следующие требования:

* структура данных – список;
* поиск – гнездовой;
* упорядочение – любой;
* корректировка – удаление записи.

1. **Описание основного варианта задания**
   1. **Структура данных**

Структуру данных по заданию предлагается реализовать с помощью списка строк, структура которого представлена на рисунке 1.

 `

**Рисунок 1 –** Структура данных

Для реализации гнездового поиска используется структура NestNode, которая состоит из следующих полей:

* буква, с которой начинаются предложения в гнезде (символ);
* указатель на следующий элемент списка с буквами;
* указатель на первый элемент вложенного списка с предложениями.

Хранение вложенных списков организовано при помощи структуры Node, которая содержит два поля:

* предложение на естественном языке (строка);
* указатель на следующий элемент списка.
  + 1. **Реализация структуры на языке C++**

Структура данных реализована на C++ при помощи класса List:

class List

{

public:

List();

std::string Find( std::string sentence );

void Sort();

void Delete( std::string sentence );

void Add( std::string sentence );

private:

struct Node

{

Node( std::string \_Sentence ) :

Sentence( \_Sentence ),

Next( nullptr ) {}

Node() :

Next( nullptr ),

Sentence() {}

Node\* Next;

std::string Sentence;

};

struct NestNode

{

NestNode( char \_Letter ) :

Letter( \_Letter ),

Next( nullptr ),

List( nullptr ) {}

char Letter;

Node\* List;

NestNode\* Next;

};

NestNode\* first;

};

* + 1. **Расчет памяти, занимаемой списком**

Объем памяти, занимаемой списком, складывается из памяти, выделяемой на внутренние поля класса, на гнезда и на элементы самого списка:

Объем памяти, занимаемой внутренними полями класса, равен указателю на первый гнездовой элемент, т.е. 4 байтам.

Объем памяти, занимаемой узлами гнезд, определяется количеством букв в алфавите естественного языка, которое умножается на сумму всех полей класса. Возьмем за основу английский алфавит, тогда объем будет таким:

Объем памяти, который расходуется на узлы списка, зависит от его полей. Предположим, что предложение имеет длину 20 символов, тогда размер элемента равен:

Тогда объем памяти, занимаемой всем списком, равен:

Таким образом, использование списка приведет к большому расходу памяти из-за использования указателей.

* 1. **Анализ метода поиска**

По условию задания необходимо использовать гнездовой метод поиска. Такой метод существенно ускоряет время поиска в списке за счет того, что поиск выполняется среди элементов одного гнезда, а не всего списка. Однако такой метод требует больших затрат памяти в сравнении с линейным поиском, так как требует хранения родительских вершин каждого гнезда.

* + 1. **Реализация метода поиска**

std::string List::Find( std::string sentence )

{

NestNode\* curNest = first;

while( curNest->Letter != sentence[0] )

{

curNest = curNest->Next;

}

Node\* cur = curNest->List;

while( cur->Sentence != sentence )

{

cur = cur->Next;

}

return cur;

}

* + 1. **Среднее количество сравнений**

Для выбора нужного гнезда:

Для поиска внутри гнезда:

Общее число сравнений:

* + 1. **Оценка времени поиска**

Поиск гнезда:

Поиск внутри гнезда:

Суммарное время поиска:

* 1. **Анализ метода упорядочивания**

Для упорядочивания использовался метод сортировки вставками. Такой метод будет хорошо работать, если на вход передается частично отсортированный массив, что нередко встречается на практике.

* + 1. **Реализация метода упорядочивания**

void List::Sort()

{

NestNode\* curNest = first;

while( curNest != nullptr )

{

Node\* cur = curNest->List;

while( cur != nullptr )

{

Node\* current = cur->Next;

std::string temp = cur->Sentence;

while( current != nullptr && current->Sentence > temp )

{

current->Next->Sentence = current->Sentence;

current = current->Next;

}

current->Sentence = temp;

cur = cur->Next;

}

curNest = curNest->Next;

}

}

* + 1. **Оценка количества сравнений**

Перебор гнезд:

Сортировка внутри гнезд:

Итоговое количество сравнений:

* + 1. **Оценка времени упорядочивания**

Перебор гнезд:

Упорядочивание элементов в гнездах:

Суммарное время сортировки:

* 1. **Метод корректировки**

Для корректировки было использовано непосредственное удаление. При удалении элемента из списка достаточно перезаписать адресное поле. Удаление гнездовой структуры при этом не осуществляется.

* + 1. **Реализация метода**

void List::Delete( std::string sentence )

{

NestNode\* curNest = first;

while( curNest->Letter != sentence[0] )

{

curNest = curNest->Next;

}

Node\* cur = curNest->List;

Node\* prev = nullptr;

while( cur->Sentence != sentence )

{

prev = cur;

cur = cur->Next;

}

if( cur == curNest->List )

{

curNest->List = cur->Next;

}

else

{

prev->Next = cur->Next;

}

delete cur;

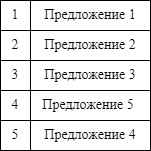
}

* + 1. **Расчет времени удаления элемента**

Время удаления будет примерно равно времени, затраченному на поиск, поскольку методы аналогичны.

1. **Альтернативный вариант**
   1. **Выбор альтернативной структуры данных**

В качестве альтернативной структуры данных был взят массив строк, представленный на рисунке 2.



**Рисунок 2 –** Альтернативная структура данных

* + 1. **Реализация структуры на языке C++**

На языке C++ структура реализована при помощи класса std::vector, который реализует массив строк.

std::vector<std::string> array;

* + 1. **Расчет занимаемой памяти**

Объем памяти, занимаемой массивом, определяется только длиной предложений, которые в нем содержатся. Таким образом, если средняя длина предложения составляет 30 символов, массив будет занимать следующий объем:

* 1. **Анализ метода поиска**

В качестве метода поиска был выбран бинарный поиск. Он требует малого количества времени и не расходует дополнительные ресурсы памяти, однако требует, чтобы поданный на вход массив уже был отсортирован.

* + 1. **Реализация метода поиска**

int Find( const std::vector<std::string>& vec, std::string target )

{

int left = 0;

int right = vec.size() - 1;

while( left <= right )

{

int mid = left + (right - left) / 2;

if( vec[mid] == target )

{

return mid;

}

else if( vec[mid] < target )

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

return -1;

}

* + 1. **Среднее количество сравнений**

Среднее количество сравнений для операции поиска равно:

* + 1. **Оценка времени поиска**
  1. **Анализ метода упорядочивания**

При использовании метода слияния исходный массив делится пополам, далее каждая часть сортируется по отдельности и производится слияние всех частей. За счет деления на части происходит ускорение работы программы.

* + 1. **Реализация метода упорядочивания**

void merge( std::vector<int>& vec, int left, int mid, int right )

{

int n1 = mid - left + 1;

int n2 = right - mid;

std::vector<int> leftSubarray( n1 );

std::vector<int> rightSubarray( n2 );

for( int i = 0; i < n1; ++i )

{

leftSubarray[i] = vec[left + i];

}

for( int j = 0; j < n2; ++j )

{

rightSubarray[j] = vec[mid + 1 + j];

}

int i = 0;

int j = 0;

int k = left;

while( i < n1 && j < n2 )

{

if( leftSubarray[i] <= rightSubarray[j] )

{

vec[k] = leftSubarray[i];

++i;

}

else

{

vec[k] = rightSubarray[j];

++j;

}

++k;

}

while( i < n1 )

{

vec[k] = leftSubarray[i];

++i;

++k;

}

while( j < n2 )

{

vec[k] = rightSubarray[j];

++j;

++k;

}

}

void partition( std::vector<int>& vec, int left, int right )

{

if( left < right )

{

int mid = left + (right - left) / 2;

partition( vec, left, mid );

partition( vec, mid + 1, right );

merge( vec, left, mid, right );

}

}

void Sort( std::vector<int>& vec )

{

partition( vec, 0, vec.size() - 1 );

}

* + 1. **Среднее количество сравнений**
    2. **Оценка объема буферных данных**

При выполнении слияния в памяти инициализируются два массива. Также при рекурсивном вызове функции разделения образуются фреймы активации, которые остаются в памяти до выхода из рекурсии.

Память, занимаемая массивами:

Память, выделяемая под рекурсию:

В итоге:

* 1. **Метод корректировки**

Для корректировки применялся метод маркировки удаляемого элемента.

* + 1. **Реализация метода**

void Delete( std::vector<std::string>& vec, std::string target )

{

int ind = Find( vec, target );

vec[ind] = "";

}

void DeleteMarked( std::vector<std::string>& vec )

{

for( int i = 0; i < vec.size(); ++i )

{

if( vec[i] == "" )

{

for( int j = i; j < vec.size() - 1; ++i )

{

vec[j] = vec[j + 1];

}

}

vec.resize( vec.size() - 1 );

}

}

* + 1. **Расчет времени удаления элемента**

Время маркировки:

Время удаления i-го элемента:

В итоге:

1. **Таблица результатов**

Таблица 1 – таблица результатов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Варианты | Структура данных | Метод поиска | Метод упорядочения | Метод корректировки |
| Основной | Список | Гнездовой | Сортировка вставками | Непосредственное удаление |
| Альтернативный | Массив | Дихотомический | Сортировка слиянием | Маркировка |

Как видно из сравнительной таблицы, к недостаткам гнездовой структуры данных можно отнести больший объем, занимаемый в памяти компьютера, а также медленную сортировку. Однако поиск элемента и его удаление реализуются быстрее, чем в массиве.

1. **Заключение**

В результате выполнения лабораторной работы были проведены качественные и количественные оценки структур данных и методов их обработки в соответствии с вариантом задания. В альтернативном варианте предложены решения, которые обеспечат более эффективные поиск, сортировку и удаление данных.