МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный исследовательский университет)

Институт №8

«Компьютерные науки и прикладная математика»

Кафедра 806

«Вычислительная математика и программирование»

Курсовой проект по дисциплине «Фундаментальные алгоритмы»

Тема: «Разработка алгоритмов системы хранения и управления данными на основе динамических структур данных»

Студент: Жураев Д.С.

Группа: М8О-213Б-21

Преподаватель: Ирбитский И.С.

Оценка:

Дата:

Москва 2023

**Содержание**

[Введение 3](#_Toc144664012)

[Аллокаторы 5](#_Toc144664013)

[Освобождение в рассортированном списке 5](#_Toc144664014)

[Освобождение с дескрипторами границ 6](#_Toc144664015)

[Система двойников 6](#_Toc144664016)

[Деревья 7](#_Toc144664017)

[Двоичное дерево поиска 7](#_Toc144664018)

[Операции двоичного дерева поиска 7](#_Toc144664019)

[Splay tree 8](#_Toc144664020)

[Операции косого дерева 9](#_Toc144664021)

[Splay 9](#_Toc144664022)

[Merge 11](#_Toc144664023)

[Поиск 12](#_Toc144664024)

[Вставка 12](#_Toc144664025)

[Удаление 12](#_Toc144664026)

[Хранилище данных 12](#_Toc144664027)

# Введение

Приложение, реализованное в рамках курсового проекта на языке программирования C++, позволяет выполнять операции над коллекциями данных заданного типа и контекстами их хранения.

Коллекция данных описывается набором строковых параметров (набор параметров однозначно идентифицирует коллекцию данных):

* название пула схем данных, хранящего схемы данных;
* название схемы данных, хранящей коллекции данных;
* название коллекции данных

Коллекция данных представляет собой ассоциативный контейнер (конкретная реализация определяется вариантом), в котором каждый объект данных соответствует некоторому уникальному ключу. Для ассоциативного контейнера необходимо вынести интерфейсную часть (в виде абстрактного класса C++) и реализовать этот интерфейс. Взаимодействие с коллекцией объектов происходит посредством выполнения одной из операций над ней:

* добавление новой записи по ключу;
* чтение записи по её ключу;
* чтение набора записей с ключами из диапазона [𝑚𝑖𝑛𝑏𝑜𝑢𝑛𝑑... 𝑚𝑎𝑥𝑏𝑜𝑢𝑛𝑑];
* обновление данных для записи по ключу;
* удаление существующей записи по ключу. Во время работы приложения возможно выполнение также следующих операций:
* добавление/удаление пулов данных;
* добавление/удаление схем данных для заданного пула данных;
* добавление/удаление коллекций данных для заданной схемы данных заданного пула данных.

Поток команд, выполняемых в рамках работы приложения, поступает из файла, путь к которому подаётся в качестве аргумента командной строки. Формат команд в файле определите самостоятельно.

Также были реализованы дополнительные задания.

Была реализована возможность кастомизации (для заданного пула схем) аллокаторов для размещения объектов данных:

* освобождение в рассортированном списке;
* освобождение с дескрипторами границ;
* система двойников.

Ассоциативный контейнер в рамках реализации данного варианта представляет собой косое дерево.

Коллекция данных представляет собой данные о встречах в рабочем календаре, содержащие следующие поля:

* id встречи (данное поле формирует уникальный ключ объекта данных);
* вид встречи (ежедневная встреча/встреча по результатам отчётного периода/собеседование/корпоратив);
* формат проведения (очный/дистанционный);
* описание встречи;
* ссылка на встречу в дистанционном формате;
* ФИО создателя встречи (раздельные поля);
* дата встречи;
* время начала встречи;
* продолжительность встречи (в минутах);
* список приглашённых на встречу (в виде строки).

# Аллокаторы

Аллокатор представляет собой объект, который управляет выделением памяти под данные, хранящиеся в коллекциях. Сам же аллокатор содержит блок памяти с сервисной информацией и блоки, которые выделяются по внешнему запросу.

Реализованы были 4 вида аллокаторов:

* на глобальной куче;
* с освобождением на рассортированном списке;
* с освобождением с дескрипторами границ;
* система двойников.

Также были реализованы методы поиска подходящего для выделения блока:

* метод первого подходящего: выбирается первый по списку свободный участок подходящего размера;
* метод лучшего подходящего: выбирается наименьший по размеру участок из всех подходящих;
* метод худшего подходящего: выбирается наибольший по размеру участок из всех подходящих.

## Освобождение в рассортированном списке

Идея данного аллокатора в том, что каждый свободный блок хранит указатель на следующий блок и свой размер, а в занятых блоках памяти хранится только размер этого блока без указателя на следующий занятый блок. Таким образом, получается рассортированный односвязный список из свободных блоков памяти.

При выделении памяти с помощью этого аллокатора возникают два случая:

* Размер блока равен размеру, запрашиваемому пользователем. В таком случае указатель предыдущего блока переставляется с текущего на следующий;
* Размер блока больше размера, запрашиваемого пользователем. В таком случае свободная часть блока отделяется от занятой, размер блока уменьшается до размера выделяемой памяти и указатель с предыдущего блока переставляется с текущего на оставшийся свободный участок.

Выделенный блок помечается как занятый и результат возвращается пользователю. Также если данный блок был первым в списке, то указатель в сервисной части аллокатора на первый свободный блок переставляется на следующий за этим.

При освобождении памяти надо найти адреса блоков, между которыми стоит наш блок. Как только мы нашли блок, адрес которого больше адреса нашего участка, то значит мы нашли освобождаемый блок. Тут также может быть два случая:

* Освобождаемый участок стоит перед следующим свободным. В таком случае мы склеиваем эти два блока, то есть увеличиваем размер, прибавляя размер этого свободного блока, и переставляем указатель на следующий свободный участок.
* Между освобождаемым и найденным участками есть еще участки. В таком случае у освобождаемого блока указатель ставится на найденный

Сложность операций выделения и освобождения памяти: O(n).

## Освобождение с дескрипторами границ

Идея данного аллокатора заключается в образовании двусязного списка из занятых участков памяти, благодаря чему освобождение происходит достаточно быстро.

Чтобы найти необходимый свободный блок при выделении, нужно вычислить разницу адресов занятых блоков. После того, как мы нашли блок, необходимо указатели у соседних блоков переставить на наш, а у нашего блока установить указатели на соседние занятые. После проверок можно вернуть данный блок пользователю.

Освобождение в данном аллокаторе достаточно простое. Необходимо найти соседние занятые блоки для нашего и переставить их указатели друг на друга. На этом освобождение заканчивается.

Сложность операции выделения: O(n).

Сложность операции освобождения: O(1).

## Система двойников

В данном аллокаторе размер кучи равен кратен целой степени двойки, то есть .

При поиске участка для выделения нужно найти блок минимального подходящего размера . Если блок найден, то используем его, в ином же случае находим ближайший блок большего размера и делим его до тех пор, пока не получим необходимый участок. Два блока, образовавшихся в результате деления называются двойниками.

Также если известен адрес и размер участка, то известен и адрес его двойника. Этот адрес можно получить с помощью операции «исключающее или».

В каждом блоке имеется дополнительный бит занятости и два указателя, указывающие на следующий и предыдущий свободные блок.

При освобождении блока памяти также может возникнуть два случая:

* Двойник свободен. В этом случае объединяются двойники до тех пор, пока это возможно.
* Двойник занят. В этом случае ищем соседние от освобождаемого блоки, затем переставляем указатели у всех блоков.

Сложность операции выделения: O(n).

Сложность операции освобождения: O().

# Деревья

## Двоичное дерево поиска

Двоичное дерево - это иерархическая структура данных, в которой каждый узел имеет значение и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне называется корнем. Узлы, не имеющие потомков, называются листьями.

Бинарное дерево поиска отличает наличие некоторых свойств: значение левого потомка меньше значения родителя, а значение правого потомка больше значения потомка родителя. Данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

### Операции двоичного дерева поиска

Поиск элемента.

Поиск элемента в двоичном дереве поиска достаточно прост, так как данные хранятся в отсортированном виде. Если искомое значение меньше значения узла, то идем в левое поддерево, в ином случае идем в правое поддерево, и так пока не найдем искомый узел.

Сложность операции поиска:

* В худшем случае: O(n);
* В среднем: O().

Вставка элемента.

Как и с поиском элемента, мы спускаемся по дереву, сравнивая значение вставляемого узла со значениями элементов дерева, и как только мы достигаем узла, у которого поддерево имеет значение NULL, вставляем туда наш узел.

Сложность операции вставки:

* В худшем случае: O(n);
* В среднем: O().

Удаление элемента.

При удалении элемента возникает три случая:

* Удаляемый элемент – лист. В этом случае просто удаляем лист из дерева.
* У удаляемого узла есть потомок. В этом случае меняем удаляемый узел на потомка, а затем удаляем наш узел.
* У удаляемого узла есть два потомка. Если в первом узле правого поддерева нет левого узла, то используем его, в другом случае находим самый левый элемент правого поддерева. После нахождения этого элемента заменяем удаляемый узел на найденный, затем удаляем наш узел.

Сложность операции удаления:

* В худшем случае: O(n);
* В среднем: O().

## Splay tree

В рамках моего варианта ассоциативный контейнер представляет собой косое дерево (splay tree), которое реализовано на основе бинарного дерева поиска.

В косом дереве, как и в АВЛ и красно-черном деревьях, поддерживается свойство сбалансированности. В splay-дереве обеспечивается выполнение операций поиска, вставки и удаления за логарифмическое время. Идея косого дерев в том, что вставка элемента, к которому недавно осуществлялся доступ, происходит в корень дерева, что позволяет получить этот элемент за O(1).

### Операции косого дерева

#### Splay

Основная операция данного дерева, которая заключается в перемещении узла в корень с помощью выполнения следующих операций:

* Zig;
* Zig-Zig;
* Zig-Zag.

Обозначим вершину, которую надо поставить на место корня, за . Родителя обозначим за , а уже его родителя за .

Операция Zig:

Также эта операция называется малый правый поворот.

Данная операция выполняется в случае, когда является корнем. Выполняется поворот по ребру между вершинами и . Эта операция нужна только для разбора крайнего случая и выполняется один раз в конце, когда изначальная глубина нечетна.

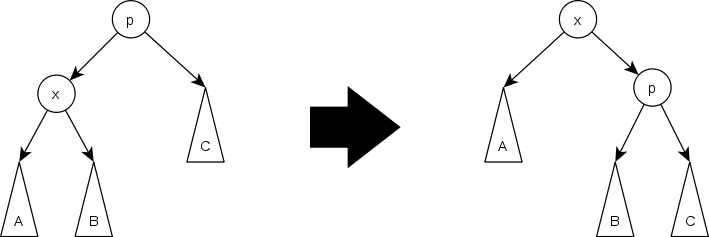


Рисунок 1. Zig (малый правый поворот)

Листинг 1. Операция Zig.

void right\_rotation(node \*&subtree\_root) const  
{  
 node \*tmp = subtree\_root;  
 subtree\_root = subtree\_root->left\_subtree\_address;  
 tmp->left\_subtree\_address = subtree\_root->right\_subtree\_address;  
 subtree\_root->right\_subtree\_address = tmp;  
}

Операция Zig-Zig:

Данная операция выполняется в случае, когда обе вершины и являются левыми сыновьями. Выполняется поворот по ребру между вершинами и , затем по ребру между вершинами и .

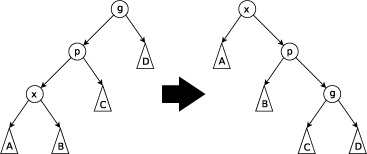


Рисунок 2. Операция Zig-Zig (большой правый поворот)

Аналогично, когда вершины и являются правыми сыновьями. В этом случае выполняется операция Zag-Zag, то есть большой левый поворот.

Операция Zig-Zag:

Данная операция выполняется в случае, когда вершина является правым потомком, а вершина – левым. Выполняется поворот по ребру между узлами и , затем по ребру между узлами и .

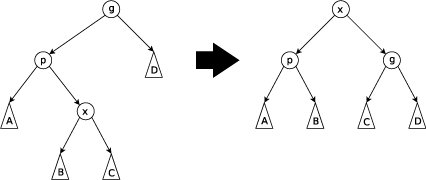


Рисунок 3. Операция Zig-Zag (малый правый поворот и малый левый поворот)

Аналогично, когда – левый потомок, а – правый. В этом случае выполняется операция Zag-Zig, то есть малый левый поворот и малый правый поворот.

Листинг 2. Операция Zag.

void left\_rotation(node \*&subtree\_root) const  
{  
 node \* tmp = subtree\_root;  
 subtree\_root = subtree\_root->right\_subtree\_address;  
 tmp->right\_subtree\_address = subtree\_root->left\_subtree\_address;  
 subtree\_root->left\_subtree\_address = tmp;  
}

#### Merge

Данная операция вызывается, когда требуется удалить узел из дерева. Эта операция нужна для слияния двух поддеревьев удаляемого узла. Сначала находим максимальный элемент поддерева, значения узлов которого больше, чем у другого поддерева, то есть у левого. Затем вызываем операцию Splay для этого узла. После этого приклеиваем правого потомка удаляемого узла к получившемуся поддереву. Таким образом мы получаем целое дерево без нарушения сбалансированности.

Листинг 3. Операция Merge.

node \* splay\_tree<tkey, tvalue, tkey\_comparer>::merge(node \*&left\_subtree, node \*&right\_subtree)  
{  
 if (left\_subtree == nullptr && right\_subtree == nullptr)  
 {  
 return nullptr;  
 }  
  
 if (left\_subtree == nullptr && right\_subtree != nullptr)  
 {  
 return right\_subtree;  
 }  
 else if (left\_subtree != nullptr && right\_subtree == nullptr)  
 {  
 return left\_subtree;  
 }  
 else  
 {  
 std::stack<node \*\*> path\_to\_subtree\_root\_exclusive;  
  
 node \* successor = left\_subtree;  
 node \*\* previous\_node\_address = &left\_subtree;  
  
 while (successor->right\_subtree\_address != nullptr)  
 {  
 path\_to\_subtree\_root\_exclusive.push(previous\_node\_address);  
 previous\_node\_address = &(successor->right\_subtree\_address);  
 successor = successor->right\_subtree\_address;  
 }  
  
 splay(successor, path\_to\_subtree\_root\_exclusive);  
  
 left\_subtree->right\_subtree\_address = right\_subtree;  
  
 return left\_subtree;  
 }  
}

#### Поиск

Поиск в косом дереве выполняется так же, как и в обычном бинарном дереве поиска, однако после завершения поиска выполняется операция Splay для найденного узла.

#### Вставка

Вставка в косом дереве похожа на вставку элемента в бинарное дерево поиска, как и операция поиска. Как и в поиске, после вставки элемента выполняется операция Splay для вставленного узла, вследствие чего узел становится корневым.

#### Удаление

При выполнении данной операции сначала находим узел, который необходимо удалить, затем вызываем операцию Splay для этого узла. После этого вызываем операцию Merge для потомков удаляемого узла.

# Хранилище данных

Хранилище представляет собой базу данных, которая хранит набор пулов. Пулы, в свою очередь, хранят схемы данных, а те хранят коллекции данных. Каждый такой ассоциативный контейнер представляет собой класс, имеющий свое имя и предоставляющий API для работы с ним.

Через входной файл с командами пользователь отправляет запрос в базу данных и получает сообщение о выполнении его запроса.