



**Министерство образования Российской Федерации
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ
им. Н.Э. БАУМАНА**

Факультет: Информатика и системы управления

Вариант №12

ДОМАШНЕЕ ЗАДАНИЕ:
По дисциплине «Алгоритмы и структуры данных»

**Студент: Сидоров А.А.
Группа: ИУ8-53**

Москва 2016

1. Структуры данных

1.1. Хэш-таблица

Краткая справка:

Идея хэширования возникла примерно в одно и то же время в разных местах мира независимо друг от друга. В январе 1953 года Г. П. Лун написал внутренний меморандум IBM, использовавший хеширование с цепочками. Примерно в это же время Джин Амдал, Элейн МакГроу, Натаниэль Рочестер и Артур Самуэль разработали программу, использующую хэширование. Разработка открытой адресации с линейным пробированием приписывается Амдалу, но у советского академика А.П.Ершова была та же идея.

Главное преимущество хэш-таблицы, по сравнению с другими табличными структурами данных, - скорость. Это преимущество оказывается более заметным, если число записей велико. Хэш-таблицы особенно эффективны, когда максимальное количество записей можно предсказать заранее, так что память под массив можно выделить один раз с оптимальным размером и впоследствии не изменять.

Также преимуществом является то, что временная сложность в среднем на операции вставки, поиска, удаления составляет $O(1)$.

Типовые варианты применения:

1. Ассоциативные массивы.

2. Индексация баз данных. Хэш-таблицы могут использоваться в качестве индексов баз данных и в качестве дисковой структуры данных, несмотря на то, что В-деревья более популярны в этом направлении. В многоузловых системах баз данных хэш-таблицы обычно используются для распределения строк между узлами, тем самым снижая сетевой трафик на хэш-соединение.

3. Кэш. При реализации кэша через хэш-таблицы коллизии могут быть обработаны путем отбрасывания одного из двух элементов, претендующих на одну и ту же ячейку; обычно стирают элемент, в

данный момент находящийся в таблице, и перезаписывают его в другую ячейку, а новый элемент записывают на бывшее место перезаписанного элемента.

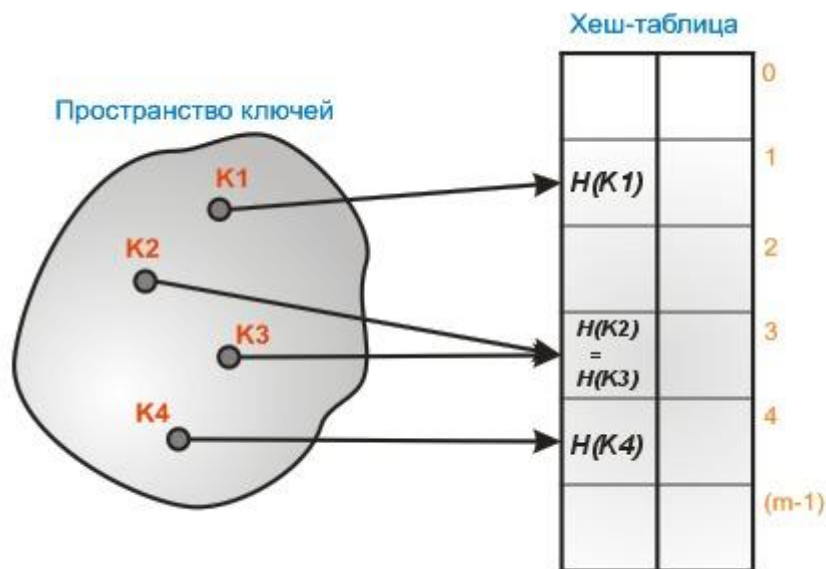
4. *Представление объектов.* Некоторые динамические языки программирования, такие как Perl, Python, JavaScript, используют хэш-таблицы для реализации объектов: ключи – это имена полей и методов, а значения – указатели на соответствующие поля и методы.

Описание:

Хэш-таблица — структура данных, реализующая интерфейс ассоциативного массива; является одним из наиболее эффективных способов реализации словаря.

Словарь – тип данных, позволяющий хранить пары ключ-значение (key-value).

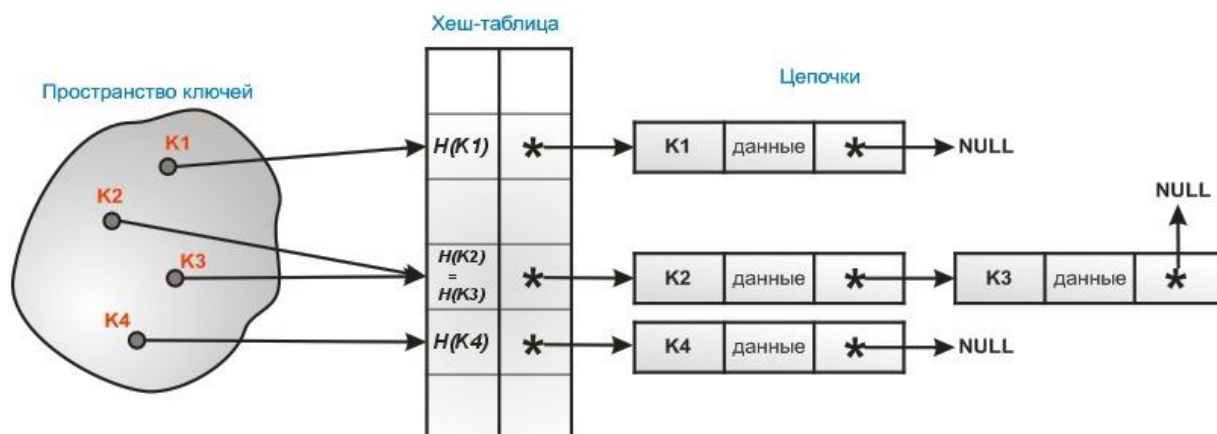
Индекс ключа в хэш-таблице определяется значением хэш-функции H от значения ключа K_i .



Как видно из рисунка размер массива хэш-таблицы равняется m . Вследствие этого может возникать ситуация, когда два значения хэш-функции от разных ключей могут определяться в одну и ту же ячейку. Такая

ситуация называется коллизией. Существует два способа разрешения коллизий: метод цепочек и открытая адресация.

В случае метода цепочек элементы, хэшированные в одну и ту же ячейку, объединяются в связный список, причем каждый новый элемент в этом списке добавляется в голову списка, т.к. операция занимает время $O(1)$.



В случае метода открытой адресации все элементы хранятся в хэш-таблице. Недостатком служит то, что может возникнуть ситуация, когда таблица полностью заполнится, и новые элементы будет некуда добавлять. Решением данной проблемы может служить динамическое увеличение размера хэш-таблицы с одновременной ее перестройкой.

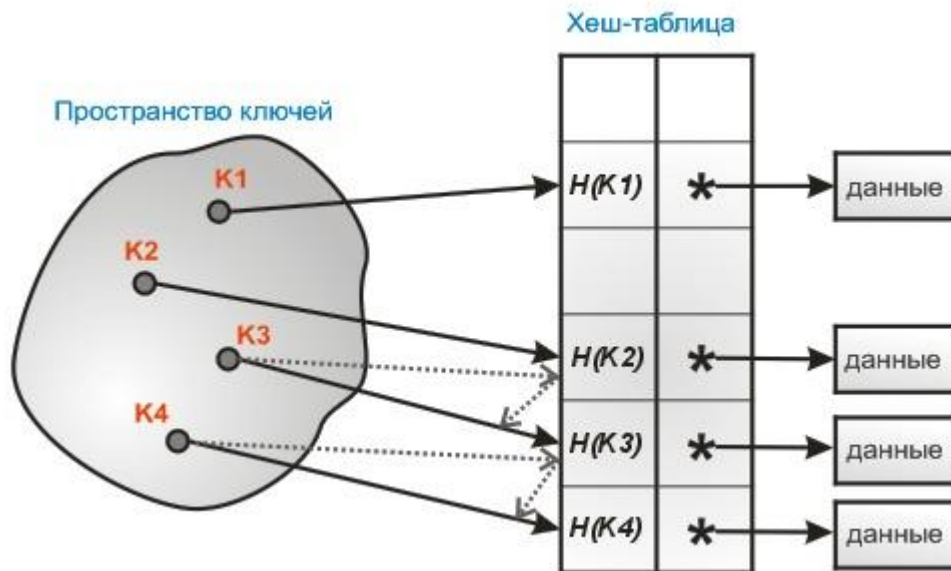
Для разрешения коллизий применяется несколько подходов: линейный, квадратичный, двойное хэширование. Индексы для элементов вычисляются следующим образом:

Линейный: $(h(x) + k \cdot i) \bmod m$;

Квадратичный: $(h(x) + k \cdot i^2 + n \cdot i) \bmod m$;

Двойное хэширование: $(h_1(x) + i \cdot h_2(x)) \bmod m$;

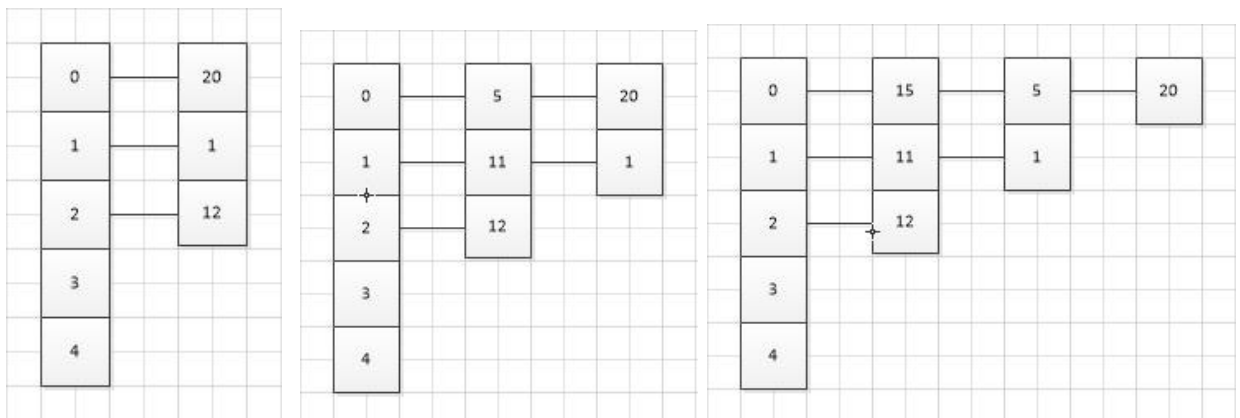
где $h(x)$, $h_1(x)$, $h_2(x)$ – функции хэширования; i – счетчик попыток (изначально равен 0, в случае, если ячейка занята, увеличивается каждый раз на единицу); m – размер хэш-таблицы; k , n – некоторые коэффициенты.



Операции в хэш-таблице с цепочками.

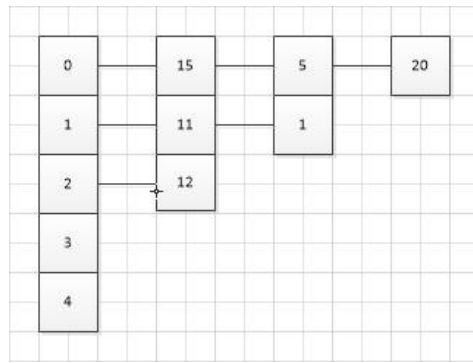
Добавление элемента: добавляем элемент k в голову списка, хранящегося в ячейке $H(k)$. Сложность $O(1)$.

Пример: добавить в таблицу элементы с ключами 20, 1, 12, 5, 11, 15; функция хэширования имеет вид $h(x) = \text{key} \bmod m$, $m = 5$.

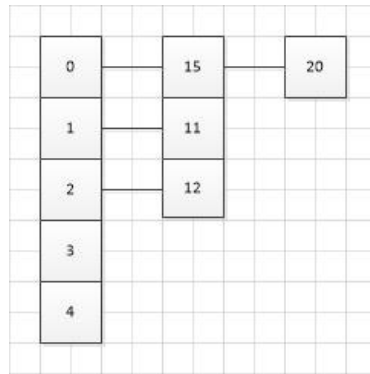


Удаление элемента: удаляем элемент k из списка, хранящегося в ячейке $H(k)$, если он там есть. Сложность $O(1 + \alpha)$, где $\alpha = n/m$ – коэффициент заполнения таблицы, n – количество элементов в таблице.

Пример: из заполненной таблицы из первого примера удалить элементы с ключом 5, 1.



До удаления:



После удаления:

Поиск элемента: ищем ключ k в списке, хранящемся в ячейке $H(k)$.

Сложность $O(1 + \alpha)$, где $\alpha = n/m$ – коэффициент заполнения таблицы, n – количество элементов в таблице.

Получение минимального/максимального элемента: ищем во всех списках хэш-таблицы минимальный/максимальный элемент. Сложность $O(n)$.

1.2. Отсортированный массив

Краткая справка:

Джон фон Нейман написал первую программу сортировки массива (сортировка слиянием) в 1945 году. В то время первая вычислительная машина с хранимой в памяти программой находилась еще на стадии разработки.

Отсортированные массивы являются наиболее пространственно-эффективной структурой данных с лучшей локализацией ссылок для последовательно хранимых данных.

Элементы в отсортированном массиве можно искать алгоритмом бинарного поиска, сложность которого $O(\log n)$, таким образом, такие массивы подходят для случаев, когда необходим быстрый поиск среди элементов. Сложность поиска для отсортированного массива такая же, как у самобалансирующихся двоичных деревьев поиска.

В некоторых структурах данных может использоваться массив структур. В таких случаях те же алгоритмы сортировки могут быть использованы и для сортировки этих структур: упорядочивание происходит по некоторому ключу, характеризующего структурный элемент. К примеру, сортировка информации о студентах по ключу в виде среднего балла.

Типовые варианты применения:

1. Коммерческое применение. Правительственным организациям, частным компаниям и интернет-приложениям приходится иметь дело с огромным числом данных. Часто требуется доступ к данным по несколько раз.

Хранение информации в отсортированном виде позволяет быстро и просто получить необходимые данные.

2. Планирование приоритетов. На уровне операционной системы множество процессов находятся в ожидании на исполнение, но процессор может обрабатывать одновременно только один процесс. Таким образом, каждому процессу ставят в соответствие приоритет. Процесс с наивысшим

приоритетом отправляется на обработку процессором. Для этих целей используется отсортированный по приоритетам массив – процесс с наивысшим приоритетом занимает первую позицию в массиве.

Описание:

Массив — структура данных в виде набора компонентов (элементов массива), расположенных в памяти непосредственно друг за другом. При этом доступ к отдельным элементам массива осуществляется с помощью индексации, то есть через ссылку на массив с указанием номера (*индекса*) нужного элемента. За счёт этого, в отличие от списка, массив является структурой данных, пригодной для осуществления произвольного доступа к её ячейкам.

В отсортированном массиве элементы идут друг за другом в порядке возрастания/убывания; поиск происходит быстрее, чем в неотсортированном массиве, но увеличивается среднее время добавления и удаления элементов.

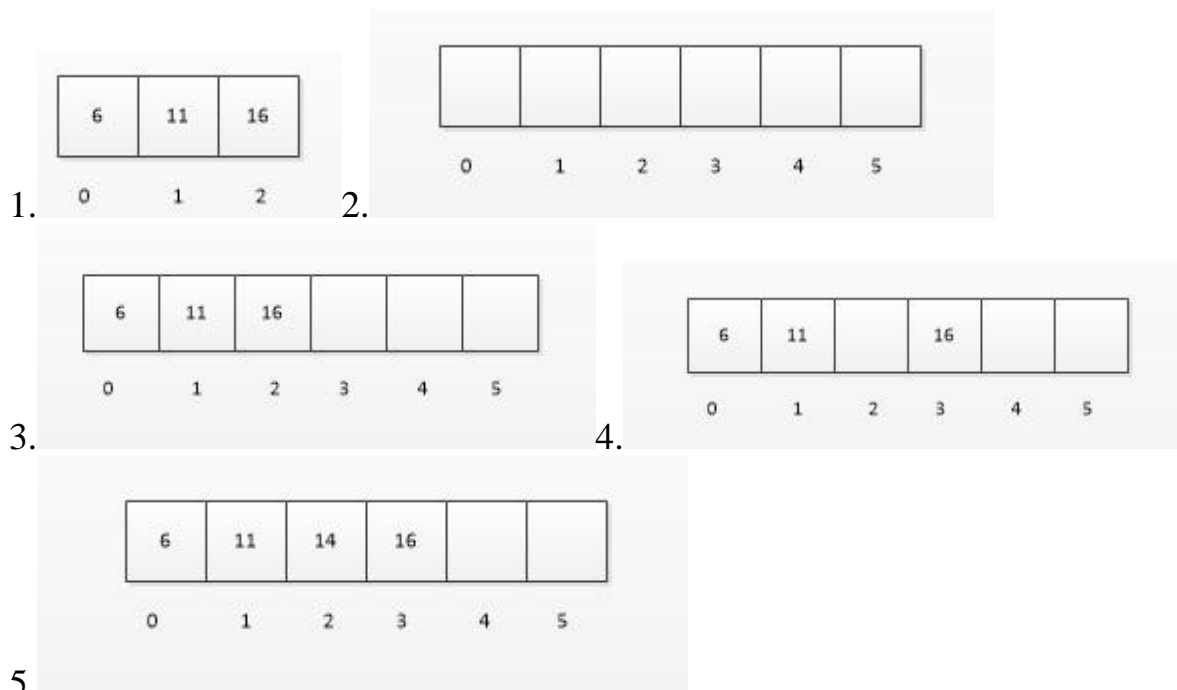
6	11	16	27	32	33	46	49	51	59
0	1	2	3	4	5	6	7	8	9

Операции в отсортированном массиве (по возрастанию):

Добавление элемента: если в массиве количество элементов равно размеру массива, выделяем новую область памяти, равную размеру предыдущего массива, умноженного на коэффициент k (обычно берут $k = 2$), копируем старый массив в новую область памяти, освобождаем старую область; находим место для нового элемента, сдвигаем оставшиеся элементы массива вправо на одну позицию, в освободившееся место вставляем новый элемент. Сложность создания нового массива $O(n)$, сложность вставки $O(n)$.

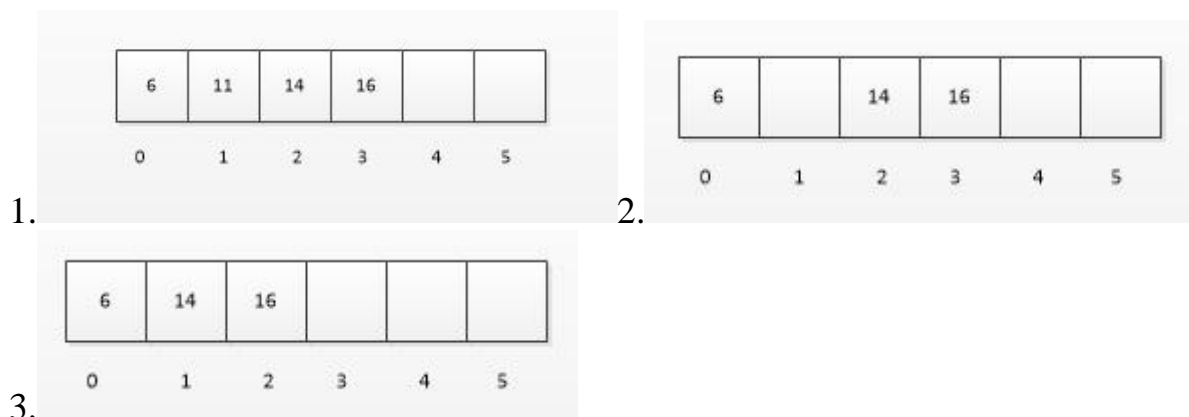
Дополнительное использование памяти $O(n)$ в случае расширения массива.

Пример: добавить в упорядоченный массив $[6, 11, 16]$ размера 3 элемент 14.



Удаление элемента: ищем элемент в массиве, если он есть, удаляем его и сдвигаем оставшуюся правую часть массива влево. Сложность $O(n)$.

Пример: удалить из массива предыдущего примера элемент 11.



Поиск элемента: ищем элемент по ключу методом бинарного поиска.

Сложность $O(\log n)$.

Получение максимального/минимального элемента: возвращаем элемент из ячейки $0/\text{size}-1$. Сложность $O(1)$.

1.3. Двоичное дерево поиска

Краткая справка:

Двоичные деревья поиска были разработаны в трудах следующих людей: Дуглас (1959), Виндли(1960), Бут(1960), Колин(1960), Хиббард(1962).

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Типовые варианты применения:

1. Сортировка. Двоичное дерево поиска может использоваться для реализации алгоритма сортировки. Схоже с пирамидальной сортировкой нужно вставлять значения, которые мы хотим отсортировать, в новую упорядоченную структуру данных – в этом случае, в двоичное дерево поиска – и затем обойти его по порядку.

2. Операции приоритетных очередей. Двоичные деревья поиска могут служить в качестве приоритетных очередей: структур, которые позволяют вставлять элемент с произвольным ключом, а также поиск и удаление элемента с максимальным (минимальным) значением ключа.

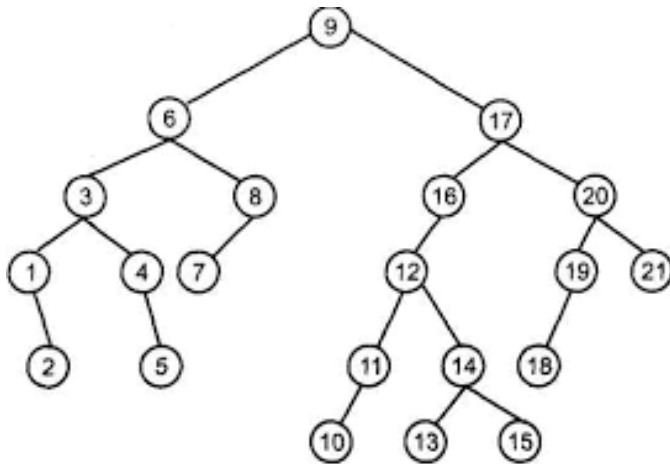
Также двоичное дерево поиска применяется для построения более абстрактных структур данных, таких как множества, мультимножества, ассоциативные массивы. Более сложные применения включают в себя ropes, различные алгоритмы вычислительной геометрии в основном в алгоритмах на основе «сканирующей прямой».

Описание:

Двоичное дерево – конечное множество вершин, которое либо пусто, либо состоит из корневой вершины, имеющей 2 ребенка – левого и правого, являющихся двоичными деревьями.

Вершина – запись, имеющая следующие поля: ключ, левый ребенок, правый ребенок, родитель, данные.

Основное свойство двоичного дерева поиска: если вершина v принадлежит левому поддереву вершины u , то $v.key < u.key$, если вершина w принадлежит правому поддереву вершины u , то $w.key \geq u.key$.



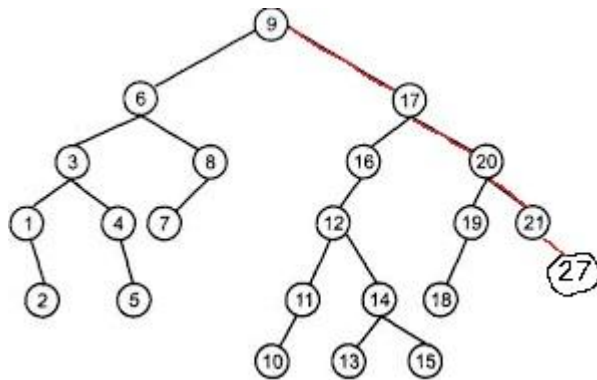
Основные операции в двоичном дереве поиска:

Добавление элемента: если дерево пустое, то добавляемый элемент объявляем корнем дерева, иначе выполняется цикл(пока не обнаружится у текущей вершины отсутствия ребенка) сравнения значения ключа добавляемого элемента со значением ключа текущей вершины:

- а) ключ добавляемого элемента больше – правого ребенка текущей вершины объявляем новой текущей вершиной;
- б) ключ добавляемого элемента меньше – левого ребенка текущей вершины объявляем новой текущей вершиной;

В случае обнаружения у текущей вершины отсутствия ребенка подвешиваем добавляемый элемент к текущей вершине в соответствии с основным свойством двоичного дерева поиска. Сложность $O(h)$, h – высота дерева.

Пример: добавить элемент 27 к дереву, изображенному в начале главы.

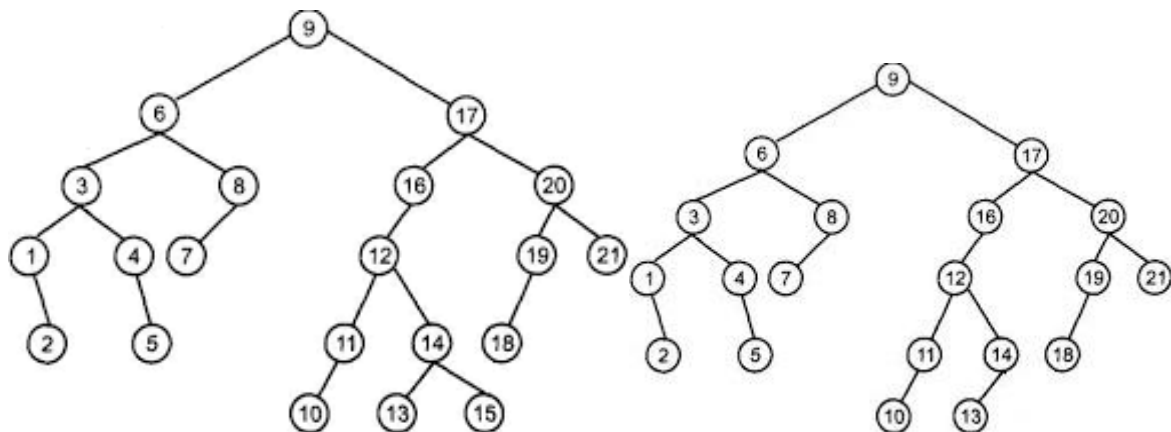


Удаление элемента: проверяем у удаляемой вершины наличие дочерних вершин:

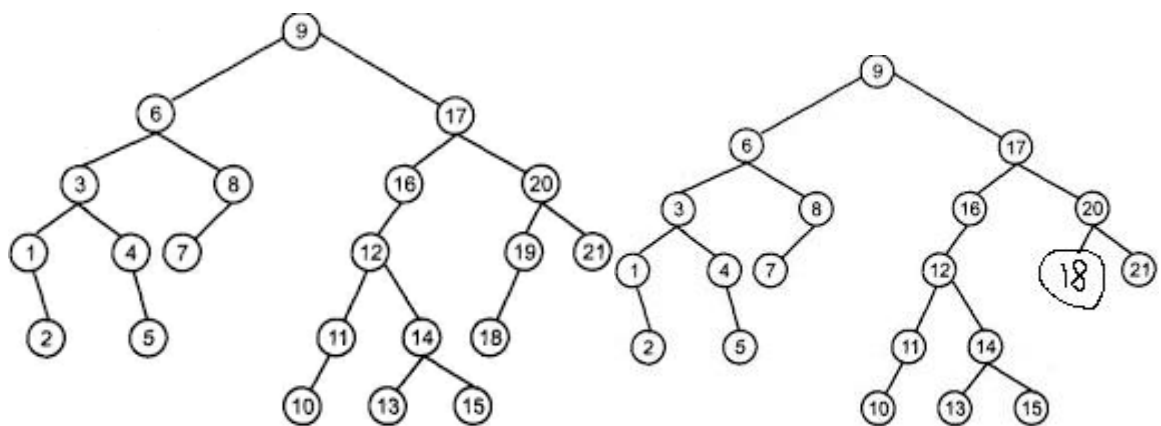
- а) присутствуют оба ребенка – ищем последующий элемент за удаляемой вершиной, удаляемую вершину заменяем найденной последующей, а правого ребенка последующей вершины (если он есть) подвешиваем на старое место родителя;
- б) присутствует один ребенок – в качестве родителя у ребенка удаляемого элемента указываем родителя удаляемого родителя, а у родителя удаляемого элемента в качестве ребенка с той же стороны указываем ребенка удаляемого элемента;
- в) у вершины нет дочерних узлов – обнуляем указатель на ребенка у родителя удаляемого элемента; Сложность $O(h)$, h – высота дерева.

Пример: удалить отдельно на каждом дереве, изображенном в начале главы, элемент 15/19/ 12.

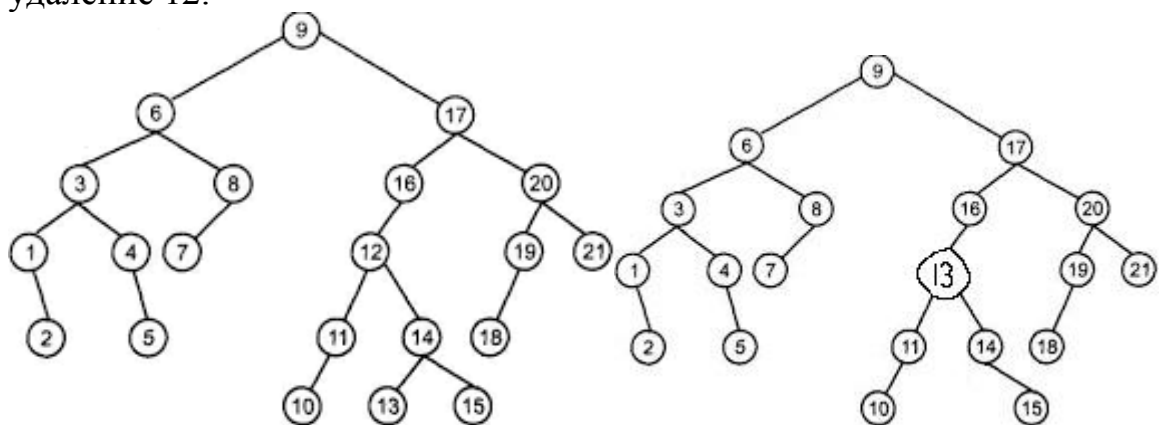
удаление 15:



удаление 19:



удаление 12:



Поиск элемента: сравниваем искомый ключ со значением ключа текущей вершины:

- а) если они совпадают или вершина отсутствует, возвращаем текущую вершину(null в случае отсутствия);
- б) если искомый ключ больше ключа текущей вершины, то вызываем рекурсивно функцию поиска для правого ребенка текущей вершины
- в) если искомый ключ меньше ключа текущей вершины, то вызываем рекурсивно функцию поиска для левого ребенка текущей вершины.

Сложность $O(h)$, h – высота дерева.

Поиск максимального/минимального элемента: опускаемся вниз по дереву по правым/левым дочерним вершинам, пока они есть; последняя вершина и будет максимальной/минимальной.

Сложность $O(h)$, h – высота дерева.

2. Описание формата входных и выходных данных.

Входными данными является набор команд, представляющих собой набор строк вида

command [key] [value]

command – исполняемая команда:

add [key] [value] – добавление элемента с заданным ключом и значением

delete [key] – удаление по ключу

search [key] – поиск по ключу

min/max – поиск минимума/максимума

print – вывод структур данных

Выходными данными является сообщение об успешном/неудачном выполнении команды, в случае успеха выводится результат и время выполнения команды.

Команда print отражает строение структур данных: хэш-таблица выводится в форме набора строк, каждая строка представляет собой ячейку хэш-таблицы и список элементов, хранящийся в ней; отсортированный массив – строка элементов; двоичное дерево поиска – последовательность элементов, соответствующая обхода дерева в ширину, у каждого элемента указан левый и правый потомок.

Входные и выходные данные находятся в файлах, пути к которым задаются через командную строку при запуске программы.

3. Тестирование

3.1 Проверка входных/выходных данных, правильности результатов работы

3.1.1 Инструкция к тестирующему ПО.

После запуска тестирующей программы необходимо ввести пути к:

1. исполняемому файлу тестируемой программы;
2. директории с набором тестов;
3. директории, в которую будут помещены файлы с результатами работы;
4. директории с файлами с верными результатами;

В результате выполнения тестирующая программа выдаст сообщение о верном/неверном выполнении тестового набора команд.

3.1.2 Описание тестов

1. проверка работы программы при некорректных входных данных;
2. проверка работы программы при добавлении элементов;
3. проверка работы программы при удалении элементов;
4. проверка работы программы при поиске элементов;
5. проверка работы программы при поиске минимума/максимума элементов;

3.2 Юнит-тестирование

Для написания юнит-тестов использовалась библиотека Google C++ Testing Framework.

3.2.1 Описание тестов

В скобках указываются названия тестов в коде тестирующего проекта.

Отсортированный массив (файл SortedArrayTest.cpp):

1. проверка корректности работы алгоритма при добавлении элемента в пустой массив (AddInEmtyArray);
2. проверка корректности работы алгоритма при добавлении элемента в начало массива(AddInBegin);

3. проверка корректности работы алгоритма при добавлении элемента в конец массива (AddInEnd);
4. проверка корректности работы алгоритма при добавлении элемента в некрайние позиции массива (AddInMid);
5. проверка корректности работы алгоритма при удалении несуществующего элемента (NotFoundToBeDeleted);
6. проверка корректности работы алгоритма при удалении из пустого массива (DeleteFromEmpty);
7. проверка корректности работы алгоритма при удалении из начала массива (DeleteFromBegin);
8. проверка корректности работы алгоритма при удалении из конца массива (DeleteFromEnd);
9. проверка корректности работы алгоритма при удалении из некрайних позиций массива (DeleteFromMid);
10. проверка корректности работы алгоритма при поиске несуществующего элемента (NotFound);
11. проверка корректности работы алгоритма при поиске в пустом массиве (SearchInEmpty);
12. проверка корректности работы алгоритма при поиске элемента, находящегося в начале массива (SearchBegin);
13. проверка корректности работы алгоритма при поиске элемента, находящегося в конце массива (SearchEnd);
14. проверка корректности работы алгоритма при поиске элемента, находящегося в некрайних позициях массива (SearchMid);
15. проверка корректности работы алгоритма при поиске минимума в пустом массиве (SearchMin);
16. проверка корректности работы алгоритма при поиске минимума в массиве (SearchMaxInEmpty);
17. проверка корректности работы алгоритма при поиске максимума в пустом массиве (SearchMaxInEmpty);

18. проверка корректности работы алгоритма при поиске максимума в массиве (SearchMax);

Хэш-таблица (файл HashTableTest.cpp):

1. проверка корректности работы алгоритма при добавлении элемента в ячейку хэш-таблицы с пустым списком (AddInEmptyList);
2. проверка корректности работы алгоритма при добавлении элемента в ячейку хэш-таблицы с непустым списком (AddTest);
3. проверка корректности работы алгоритма при удалении элемента из ячейки хэш-таблицы с пустым списком (DeleteFromEmptyList);
4. проверка корректности работы алгоритма при удалении несуществующего элемента (NotFoundToBeDeleted);
5. проверка корректности работы алгоритма при удалении элемента, находящегося в голове списка (DeleteListHead);
6. проверка корректности работы алгоритма при удалении элемента, находящегося в конце списка (DeleteListEnd);
7. проверка корректности работы алгоритма при удалении элемента, находящегося в крайних позициях списка (DeleteListEnd);
8. проверка корректности работы алгоритма при поиске несуществующего элемента (NotFound);
9. проверка корректности работы алгоритма при поиске элемента в ячейке хэш-таблицы с пустым списком (SearchInEmptyList);
10. проверка корректности работы алгоритма при поиске элемента, находящегося в голове списка (SearchInBeginList);
11. проверка корректности работы алгоритма при поиске элемента, находящегося в конце списка (SearchInEndList);
12. проверка корректности работы алгоритма при поиске элемента, находящегося в крайних позициях списка (SearchInMidList);
13. проверка корректности работы алгоритма при поиске минимума в пустой хэш-таблице (SearchMinInEmptyTable);

14. проверка корректности работы алгоритма при поиске максимума в пустой хэш-таблице (SearchMaxInEmptyTable);
15. проверка корректности работы алгоритма при поиске минимума в хэш-таблице (SearchMin);
16. проверка корректности работы алгоритма при поиске максимума в хэш-таблице (SearchMax);

Двоичное дерево поиска (файл BinarySearchTreeTest.cpp):

1. проверка корректности работы алгоритма при добавлении элемента в пустое дерево (AddInEmptyTree);
2. проверка корректности работы алгоритма при добавлении минимального элемента (AddLeftTest);
3. проверка корректности работы алгоритма при добавлении максимального элемента (AddRightTest);
4. проверка корректности работы алгоритма при добавлении элемента, не являющегося минимумом и максимум (AddMidTest);
5. проверка корректности работы алгоритма при удалении из пустого дерева (DeleteFromEmpty);
6. проверка корректности работы алгоритма при удалении несуществующего элемента (NotFoundToBeDeleted);
7. проверка корректности работы алгоритма при удалении элемента без потомков (DeleteWithNoChildren);
8. проверка корректности работы алгоритма при удалении элемента с одним потомком (DeleteWithOneChild);
9. проверка корректности работы алгоритма при удалении элемента с двумя потомками (DeleteWithTwoChildren);
10. проверка корректности работы алгоритма при поиске несуществующего элемента (NotFound);
11. проверка корректности работы алгоритма при поиске в пустом дереве (SearchInEmpty);

12. проверка корректности работы алгоритма при поиске элемента, находящегося в самом низу дерева (SearchLower);
13. проверка корректности работы алгоритма при поиске элемента, являющегося корнем дерева (SearchRoot);
14. проверка корректности работы алгоритма при поиске элемента, имеющего одного или двух потомков (SearchMid);
15. проверка корректности работы алгоритма при поиске минимума в пустом дереве (MinInEmpty);
16. проверка корректности работы алгоритма при поиске максимума в пустом дереве (MaxInEmpty);
17. проверка корректности работы алгоритма при поиске минимума (SearchMin);
18. проверка корректности работы алгоритма при поиске максимума (SearchMax);