ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных» на тему

ДЕРЕВО ТУРНИРОВ (TOURNAMENT TREE, WINNER TREE)

|  |  |
| --- | --- |
| Выполнил студент | Пухов М.С. |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИС242 |

|  |  |  |
| --- | --- | --- |
| Работу приняла |  | ассистент Кафедры ВС Насонова А.О. |
| подпись | | |

|  |  |  |
| --- | --- | --- |
| Защищена |  | Оценка |

Новосибирск – 2023

# Оглавление

[ВВЕДЕНИЕ 3](#_Toc153900010)

[ОПИСАНИЕ СТРУКТУРЫ 4](#_Toc153900011)

[ВИДЫ РЕАЛИЗАЦИЙ ДЕРЕВА ТУРНИРОВ 6](#_Toc153900012)

[Основная реализация Tournament sort 6](#_Toc153900013)

[Реализация с многопутевым слиянием 11](#_Toc153900014)

[ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ 15](#_Toc153900015)

[1. Вставка ключа в дерево 15](#_Toc153900016)

[2. Удаление ключа из дерева 17](#_Toc153900018)

[3. Поиск 19](#_Toc153900030)

[АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ 21](#_Toc153900039)

[1. Вставка ключа в дерево 21](#_Toc153900040)

[2. Удаление ключа из дерева 22](#_Toc153900041)

[3. Поиск 25](#_Toc153900042)

[ЗАКЛЮЧЕНИЕ 27](#_Toc153900043)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 28](#_Toc153900044)

[ПРИЛОЖЕНИЕ 29](#_Toc153900045)

# ВВЕДЕНИЕ

Дерево турниров, также известное как "tournament tree" или "winner tree", представляет собой структуру данных, широко применяемую в информатике для эффективного поиска максимального (или минимального) элемента в массиве данных. Эта структура, хотя и кажется простой, обладает высокой степенью эффективности, что делает ее ключевым элементом многих алгоритмов, особенно в области сортировки и поиска.

Идея дерева турниров уходит корнями в исследования по алгоритмам сортировки и поиску, проведенные в середине XX века. Она была впервые формализована в работах выдающихся ученых, таких как Дональд Кнут, в его трудах по анализу алгоритмов. Термин "дерево турниров" стал широко распространен в литературе с середины 1970-х годов.

Дерево турниров нашло свое место в различных областях компьютерных наук и информационных технологий. Одним из ключевых применений является его использование в алгоритмах сортировки, таких как турнирная сортировка (tournament sort), где эффективность поиска максимального элемента играет решающую роль. Также оно активно применяется в структурах данных, используемых в сетевых технологиях, для оптимизации маршрутизации и управления данными.

Таким образом, данное исследование посвящено более глубокому пониманию принципов работы дерева турниров, его историческому контексту и практическим применениям, предоставляя полезные инсайты для разработчиков и исследователей в области алгоритмов и структур данных.

# ОПИСАНИЕ СТРУКТУРЫ

Дерево турниров представляет из себя бинарное дерево, где каждый узел содержит информацию о турнире между двумя элементами. В этом контексте, турнир - это сравнение двух элементов с последующим определением победителя.

Структурные элементы дерева турниров:

* Листья (листовые узлы): Каждый лист содержит один элемент массива данных. Листья представляют собой начальные участники "турнира" - сравнения двух элементов.
* Промежуточные узлы: Эти узлы представляют собой результаты турниров между листьями или другими промежуточными узлами. Каждый промежуточный узел хранит информацию о победителе в соответствующем турнире.

Структура дерева турниров позволяет эффективно определить максимальный (или минимальный) элемент в массиве данных. Корень дерева содержит общего победителя, представляя собой максимальный элемент, если дерево используется для поиска максимума.

Основные шаги алгоритма дерева турниров:

* Инициализация: Заполняем листовые узлы дерева элементами из массива данных. Каждый лист - это участник "турнира".
* Проход вверх по дереву: Начинаем сравнивать элементы в каждом турнире и определять победителя. Победитель сохраняется в соответствующем промежуточном узле.
* Движение к корню: Процесс продолжается вверх по дереву, определяя победителей в каждом уровне, пока не достигнем корня. В результате корень содержит общего победителя, представляющего максимальный (или минимальный) элемент.
* Обновление данных: При изменении элемента массива, например, при сортировке, пересчитываем победителей в соответствующих турнирах и обновляем структуру дерева.

Особенности и преимущества:

* Временная сложность: Операция поиска максимума выполняется за время O(log n), где n - количество элементов в массиве.
* Обновление: Дерево турниров легко обновляется при изменении элементов массива, что обеспечивает эффективность в динамических сценариях.
* Применение: Находит широкое применение в алгоритмах сортировки, поиске и управлении данными в сетях.

Понимание внутренней структуры дерева турниров является ключевым аспектом для оптимизации алгоритмов, использующих данную структуру данных. Разделение на листья и промежуточные узлы, эффективность операций и универсальность применения делают дерево турниров важным инструментом в инженерии алгоритмов.

# ВИДЫ РЕАЛИЗАЦИЙ ДЕРЕВА ТУРНИРОВ

## Основная реализация Tournament sort

Турнирная сортировка — это алгоритм сортировки, который оперирует с понятием "турнира". Процесс деления на победителей и проигравших осуществляется следующим образом:

1. Формирование турнира (Рисунок 1):

* Исходный массив элементов рассматривается как листья бинарного дерева (турнира).
  + Пары элементов сравниваются, и из каждой пары выбирается наименьший элемент, который становится победителем в данной паре. Таким образом, на верхнем уровне дерева формируется новый массив, состоящий из победителей пар.

1. Повторение (Рисунок 2):
   * Процесс выбора победителей продолжается до тех пор, пока не определится финальный победитель – наименьший элемент в массиве. Этот элемент будет находиться в корне бинарного дерева.
2. Деление (Рисунок 3):
   * Массив разделяется на две части: одну с победителями (правая подветвь), вторую – с проигравшими (левая подветвь). Таким образом, элемент в корне становится первым элементом отсортированного массива.
3. Рекурсия (Рисунок 4):
   * Рекурсивно повторяется процесс для обеих частей (правой и левой подветвей). Каждая из этих частей рассматривается как отдельный турнир, и процесс сравнения и выбора победителей повторяется.
4. Завершение (Рисунок 5):
   * Рекурсивные вызовы продолжаются, пока не будет достигнута базовая единица (одиночный элемент) в каждой подветви. Тогда весь массив будет отсортирован.

Таким образом, турнирная сортировка разделяет массив на подмассивы, определяет победителей и проигравших, а затем рекурсивно повторяет этот процесс до полной сортировки.

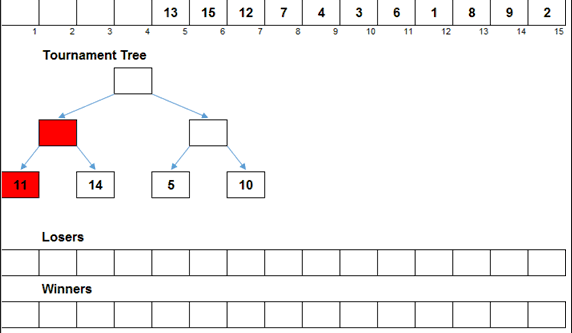


Рисунок 1 (а). Начинается сравнение элементов на нижнем уровне.

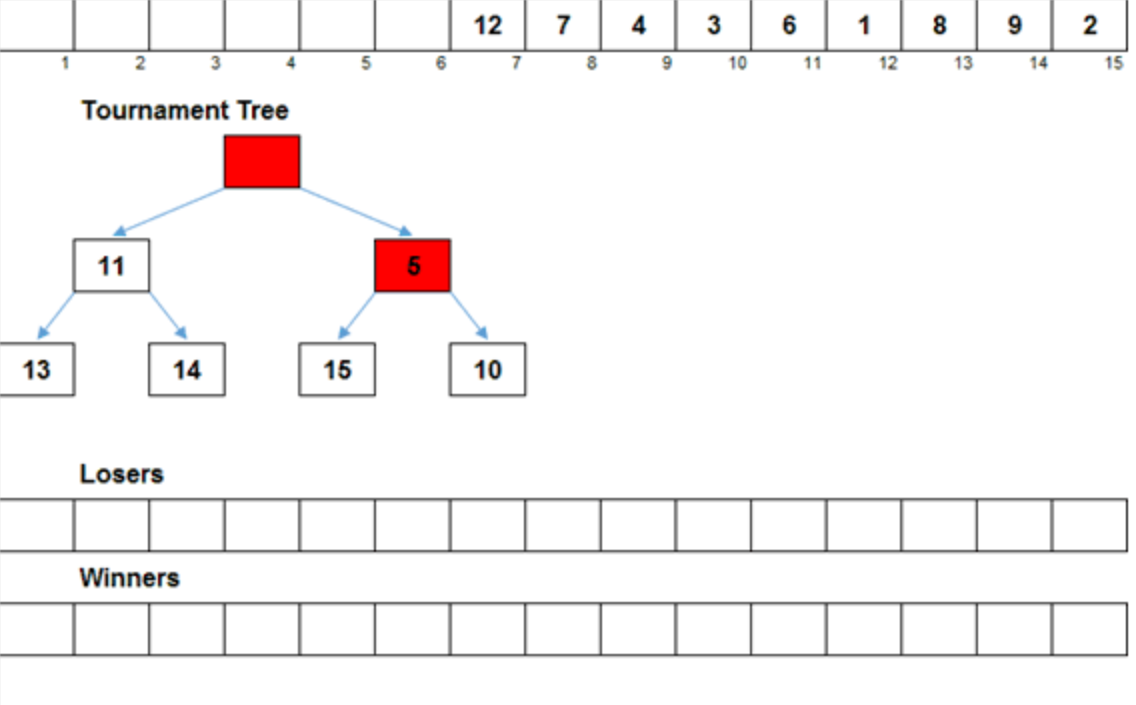


Рисунок 1 (б). Наименьший элемент поднимается на уровень выше.

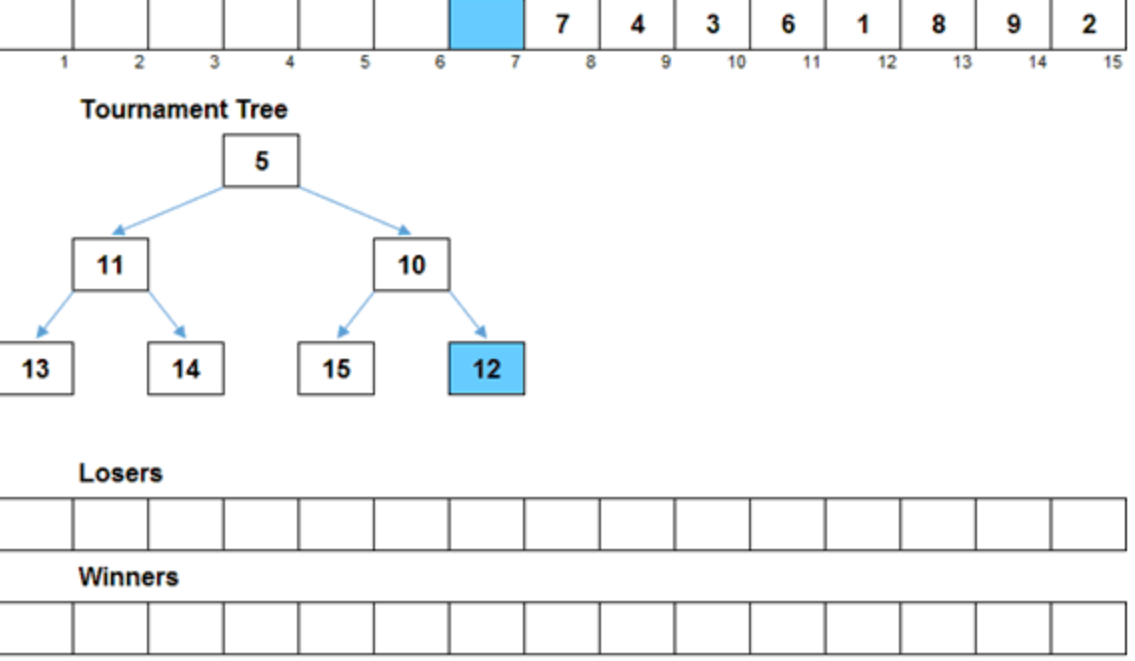


Рисунок 2 (а). Свободная ячейка заполняется следующим элементом массива.

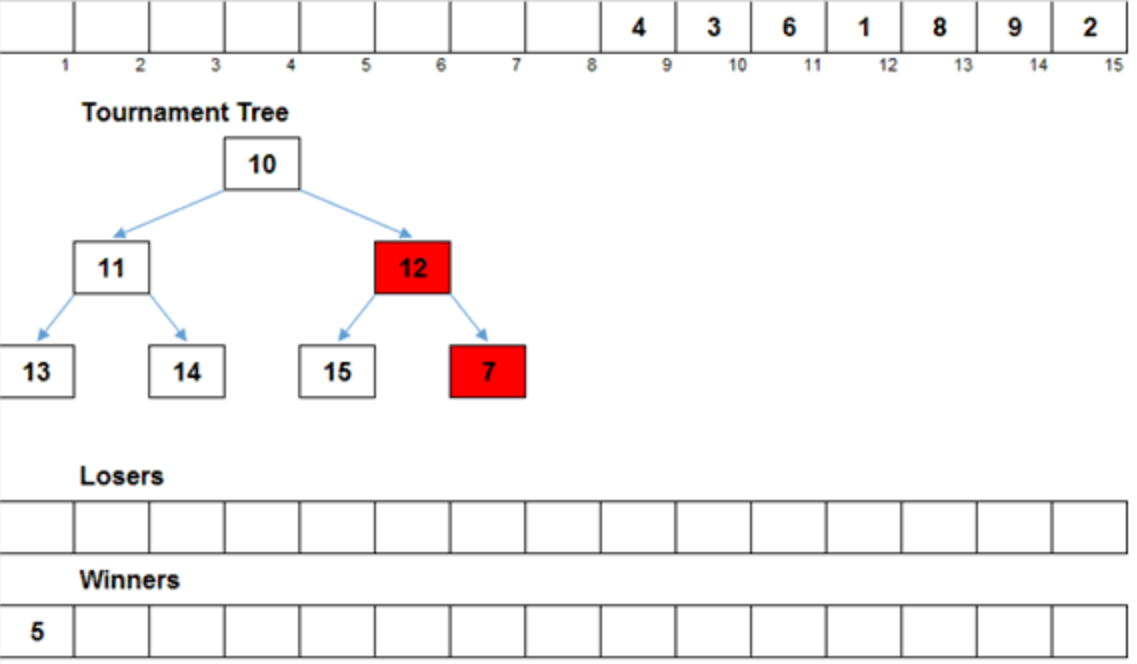


Рисунок 2 (б). Происходит сравнение нового элемента (сначала в паре, затем с элементом высшего уровня).

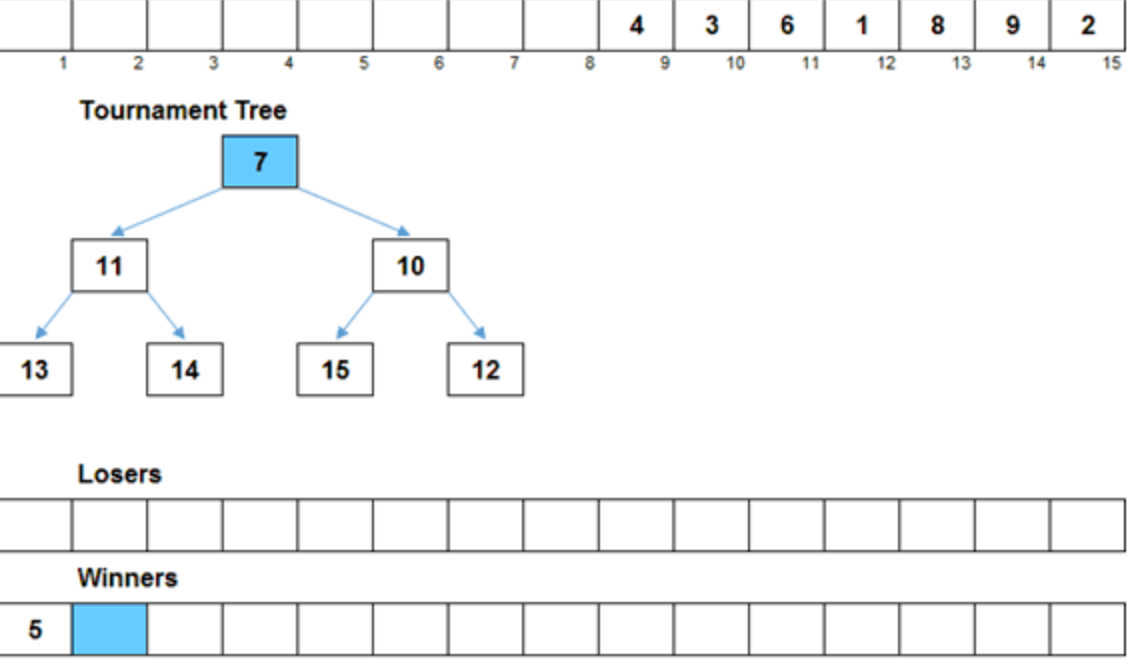


Рисунок 3 (а). Определённый победитель с верхнего уровня дерева перемещается в массив победителей.

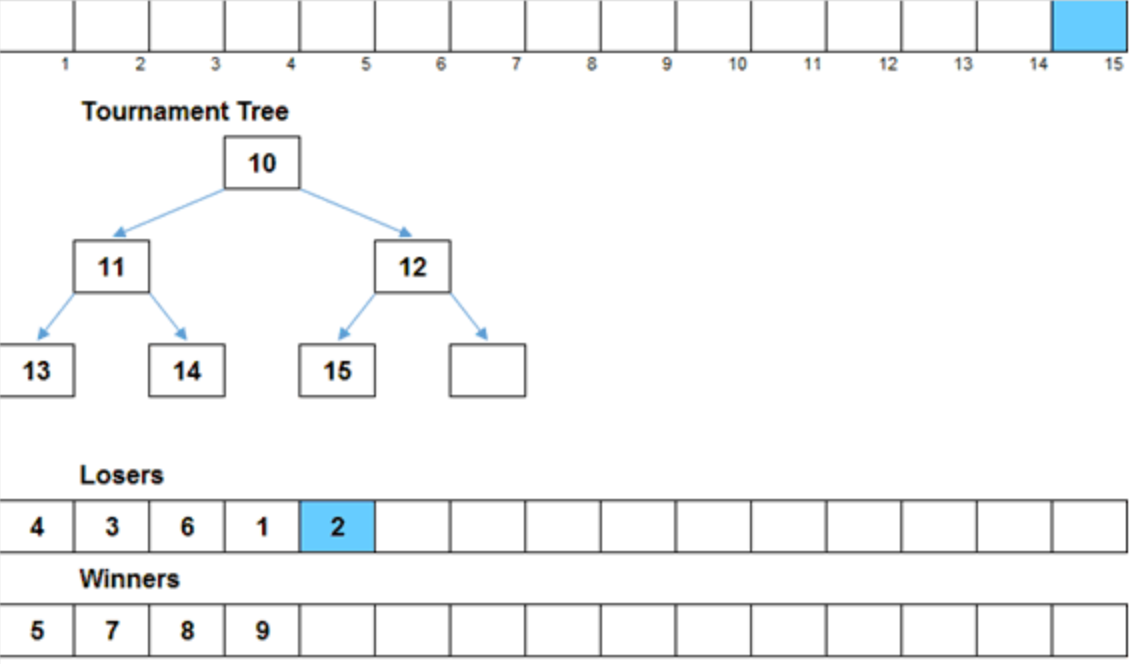


Рисунок 3 (б). Меньшие элементы перемещаются в массив проиграввших.

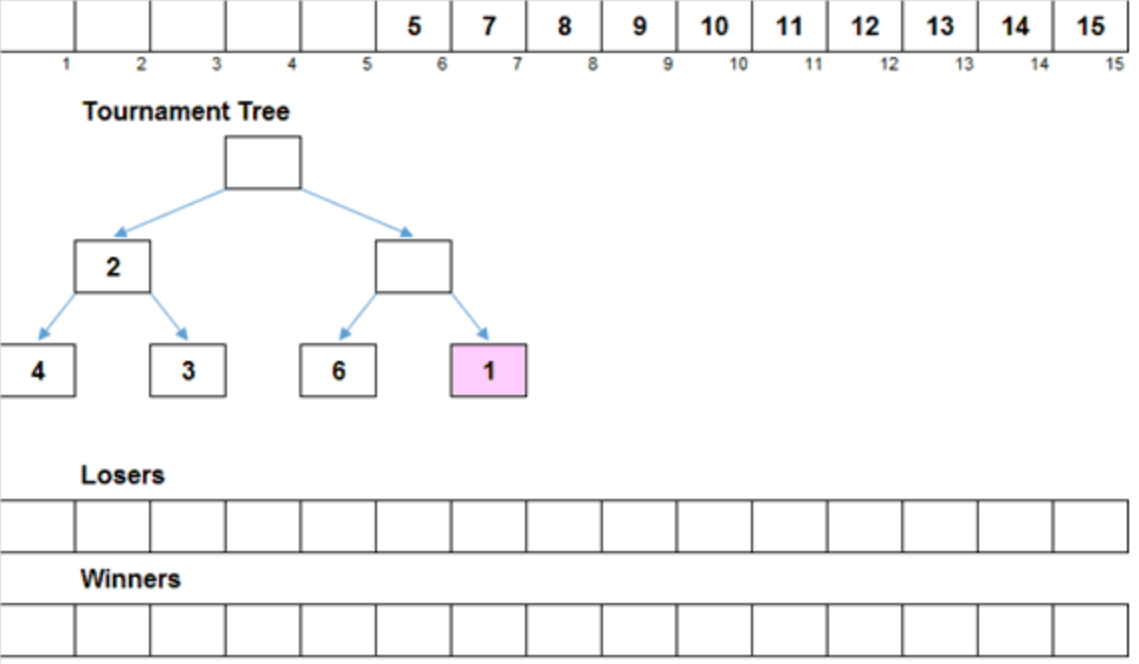


Рисунок 4. Дерево снова заполняется элементами: сначала из массива проигравших, затем из массива победителей.

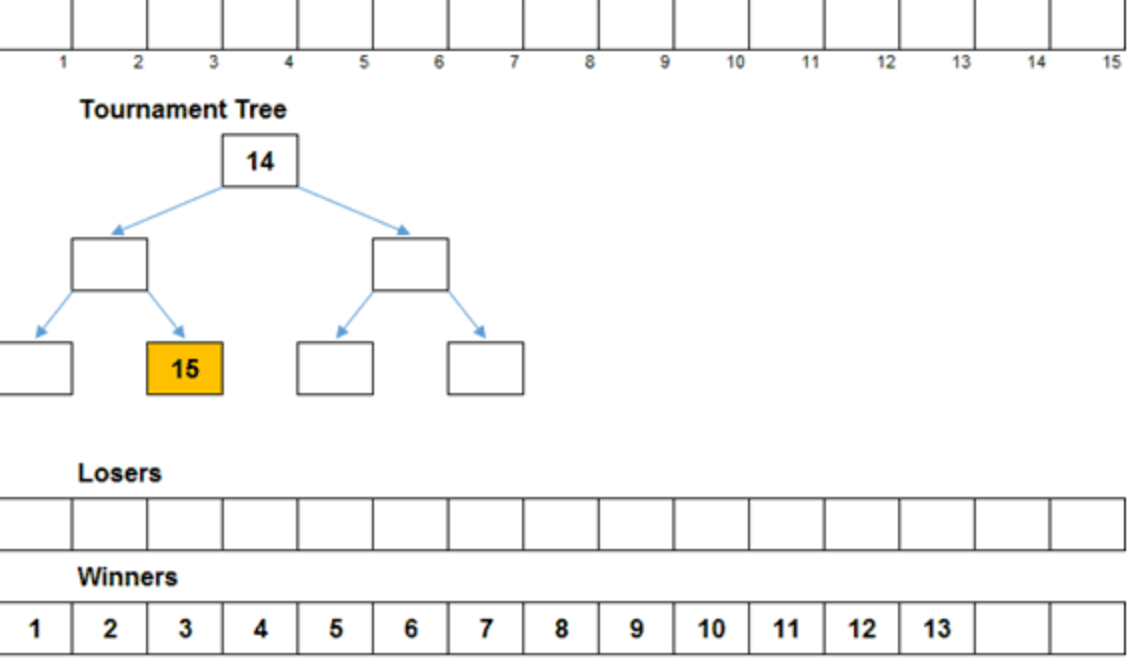


Рисунок 5 (а). Заполняется конечный массив.

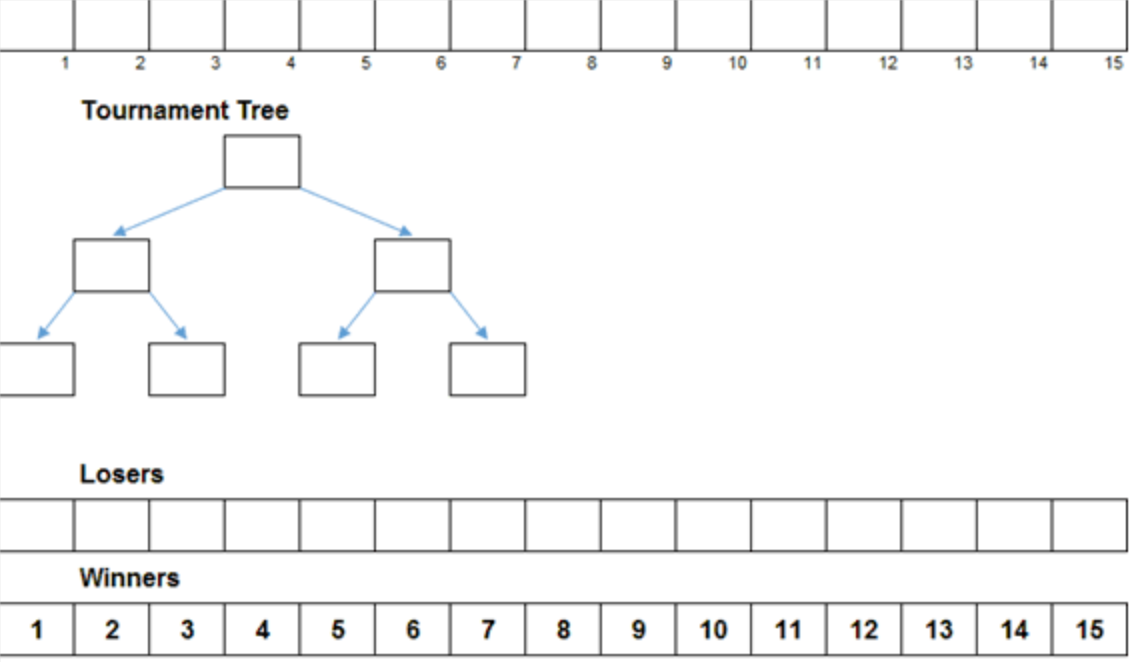


Рисунок 5 (б). Завершение сортировки.

Это наивная реализация, в которой после каждого разделения на «проигравших» и «победителей» эти массивы объединяются, и затем для объединённого массива все действия повторяются снова. Если в объединённом массиве будут сначала идти «проигравшие» элементы, то просеивание через турнирную кучу упорядочит их совместно с «победителями».

Данная реализация достаточно проста и наглядна, но эффективной она не является. Отсортированные «победители» проходят через турнирное дерево неоднократно, что выглядит, очевидно, излишним. Временна́я сложность здесь логарифмично-квадратичная - O(n \* log2n).

## Реализация с многопутевым слиянием

Алгоритм заметно ускоряется, если прошедшие через сито упорядоченные элементы не пропускать через турнирные забеги повторно.

Начинается сортировка по уже описанному методу (Рисунок 1). После того как неупорядоченный массив разделится на отсортированных «победителей» и неотсортированных «проигравших» (Рисунок 2), весь процесс заново повторяем только для «проигравших» (Рисунок 3). На каждой новой итерации будет накапливаться набор массивов с «победителями» (Рисунок 4) и так будет продолжаться до тех пор, пока на каком-то шаге «проигравших» уже не останется. После чего останется только произвести слияние всех массивов «победителей» (Рисунок 5). Так как все эти массивы отсортированы, это слияние проходит с минимальными затратами.

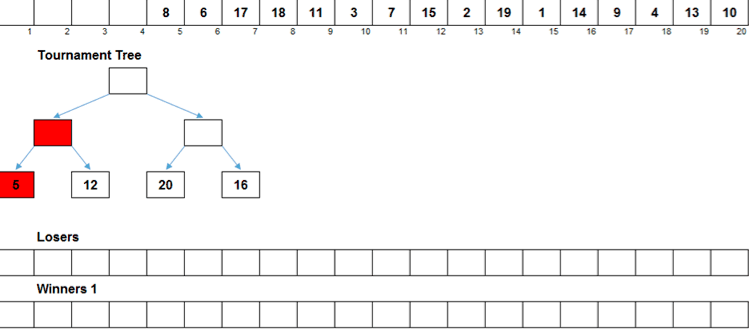


Рисунок 1. Происходит сравнение нового элемента (сначала в паре, затем с элементом высшего уровня).

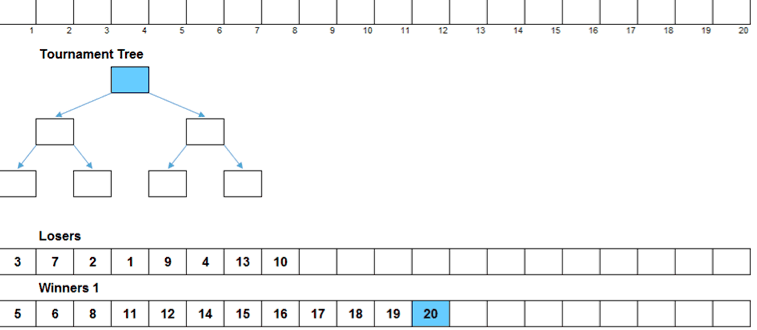


Рисунок 2. Определившиеся победители перемещается в массив победителей, а меньшие элементы перемещаются в массив проигравших.

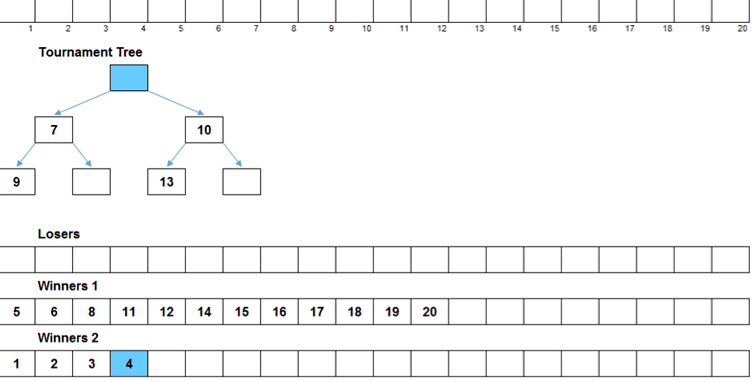


Рисунок 3. Для проигравших процесс повторяется. Новые победители заполняют массив 2.

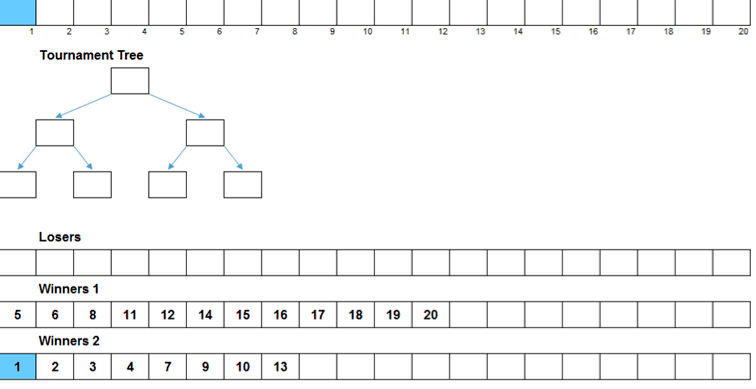


Рисунок 4. По завершении сортировки имеем 2 массива победителей.

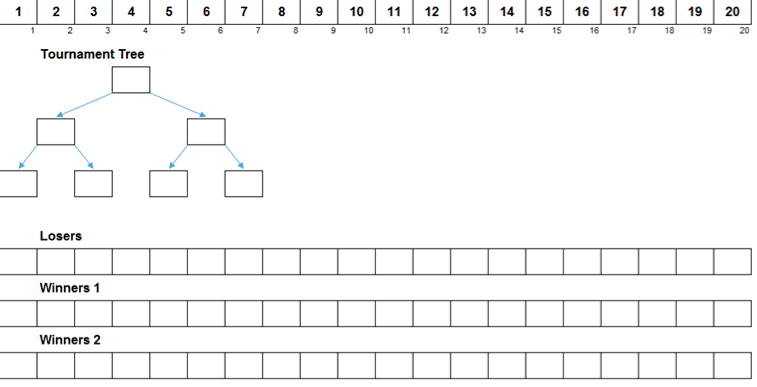


Рисунок 5. Объединение двух массивов, начиная со второго.

В таком виде алгоритм может уже вполне найти полезное применение. Например, если приходится работать с big data, то по ходу процесса с помощью турнирной кучи будут выходить пакеты упорядоченных данных, с которыми уже можно что-нибудь делать, пока обрабатывается всё остальное.

Каждый из n элементов массива проходит через турнирное дерево всего один раз, что обходится в O(log n) по времени. В такой реализации алгоритм имеет худшую/среднюю временну́ю сложность O(n \* log n).

# ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ

## Вставка ключа в дерево

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **void** **insertElement**(**int** \*array, **int** \*\*tree, **int** left, **int** right, **int** position, **int** value, **int** \*n) {  // Смещаем элементы вправо, чтобы освободить место для нового элемента  **for** (**int** i = \*n; i > position; --i) {  array[i] = array[i - **1**];  }  array[position] = value; // Вставляем новый элемент  (\*n)++; // Увеличиваем размер массива  // Перевыделяем память под дерево турниров  **int** height = (**int**)ceil(log2(\*n));  **int** maxSize = **2** \* (**int**)pow(**2**, height) - **1**;  \*tree = realloc(\*tree, maxSize \* **sizeof**(**int**));  // Перестраиваем дерево  buildTournamentTree(array, \*tree, **0**, **0**, \*n - **1**);  } |

Цель данной функции - обеспечить эффективную вставку нового элемента в бинарное турнирное дерево, представленное массивом. Весь процесс разделен на несколько шагов для эффективного управления данными и поддержания структуры дерева.  
  
1. Смещение элементов:  
- Итеративно смещаем элементы массива вправо, начиная с конечной позиции и заканчивая позицией вставки нового элемента. Это необходимо для создания места под новый элемент.  
  
2. Вставка нового элемента:  
- Вставляем новый элемент в массив на указанную позицию. Этот шаг обеспечивает корректное внесение нового значения в структуру данных.  
  
3. Увеличение размера массива:  
- Увеличиваем размер массива на 1, поскольку вставили новый элемент. Это важно для учета изменений в структуре данных.  
  
4. Перевыделение памяти для дерева турниров:  
- Вычисляем высоту дерева с использованием логарифмической функции и определяем максимальный размер, который может занять дерево с учетом этой высоты.  
- Перевыделяем память под дерево, используя функцию `realloc`, чтобы учесть возможное увеличение размера.  
  
5. Перестройка дерева турниров:  
- Вызываем функцию `buildTournamentTree`, которая перестраивает бинарное турнирное дерево, учитывая изменения в массиве после вставки нового элемента. Этот шаг гарантирует корректную структуру дерева и его согласованность с данными массива.  
  
В результате этих шагов функция позволяет безопасно вставить новый элемент в дерево турниров, поддерживая при этом его упорядоченность и структуру.

## Удаление ключа из дерева

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **void** **deleteElement**(**int** \*array, **int** \*\*tree, **int** left, **int** right, **int** position, **int** \*n) {  // Элемент исчезает из массива, смещаем остальные элементы  **for** (**int** i = position; i < \*n - **1**; ++i) {  array[i] = array[i + **1**];  }  (\*n)--; // Уменьшаем размер массива  // Перевыделяем память под дерево турниров  **int** height = (**int**)ceil(log2(\*n));  **int** maxSize = **2** \* (**int**)pow(**2**, height) - **1**;  \*tree = realloc(\*tree, maxSize \* **sizeof**(**int**));  // Перестраиваем дерево  buildTournamentTree(array, \*tree, **0**, **0**, \*n - **1**);  } |

Цель данной функции заключается в обеспечении корректного удаления элемента из бинарного турнирного дерева, представленного массивом. Процесс удаления включает несколько этапов для сохранения структуры дерева и поддержания корректности данных.

1. Исчезновение элемента из массива:

- Итеративно перемещаем элементы массива влево, начиная с позиции удаления и заканчивая концом массива. Это действие осуществляется для заполнения места, оставшегося после удаления элемента.

2. Уменьшение размера массива:

- Уменьшаем размер массива на 1, так как один элемент был удален. Это важно для актуализации размера структуры данных.

3. Перевыделение памяти для дерева турниров:

- Рассчитываем высоту дерева с использованием логарифмической функции и определяем максимальный размер, который может занять дерево с учетом нового размера массива.

- Перевыделяем память под дерево, используя функцию `realloc`, чтобы учесть изменения размера.

4. Перестройка дерева турниров:

- Вызываем функцию `buildTournamentTree`, которая перестраивает бинарное турнирное дерево, учитывая изменения в массиве после удаления элемента. Этот шаг гарантирует корректную структуру дерева и его согласованность с данными массива.

В результате этих операций функция обеспечивает безопасное удаление элемента из дерева турниров, сохраняя его структуру и упорядоченность.

## Поиск

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **int** **findIndexInArray**(**int** \*array, **int** n, **int** value) {  **for** (**int** i = **0**; i < n; i++) {  **if** (array[i] == value) {  **return** i; // Возвращаем индекс элемента в исходном массиве  }  }  **return** -**1**; // Элемент не найден  } |

Цель данной функции заключается в выполнении поиска заданного элемента в массиве. Процесс поиска организован в цикле, где каждый элемент массива сравнивается с целевым значением.

1. Итерация по массиву:

- Функция использует цикл для итерации по каждому элементу массива, начиная с первого и заканчивая последним.

2. Сравнение элементов:

- На каждом шаге цикла сравниваем текущий элемент с целевым значением. Если совпадение найдено, функция возвращает индекс этого элемента в исходном массиве.

3. Возвращение результата:

- Если элемент не найден после завершения цикла, функция возвращает значение -1. Это служит индикатором того, что искомый элемент отсутствует в массиве.

В результате выполнения этой функции, если элемент найден, возвращается его индекс; в противном случае, возвращается -1, указывая на отсутствие искомого элемента в массиве. Данная процедура является ключевым этапом для реализации операций поиска в вашей программе, обеспечивая эффективный доступ к данным.

# АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ

## Вставка ключа в дерево

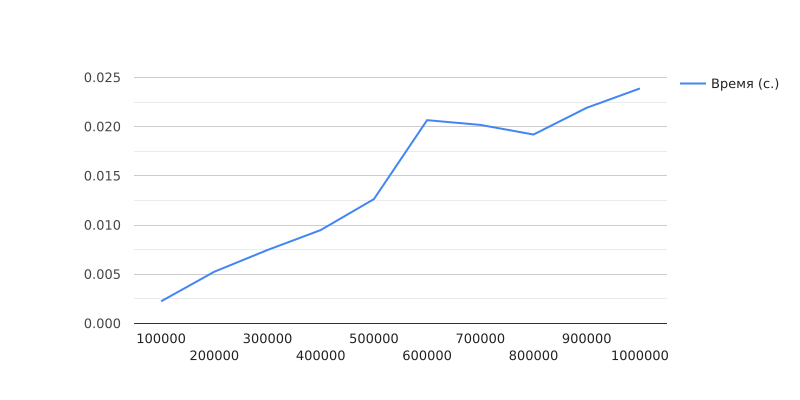


Рисунок 1 (а). Вставка ключа в упорядоченное дерево.

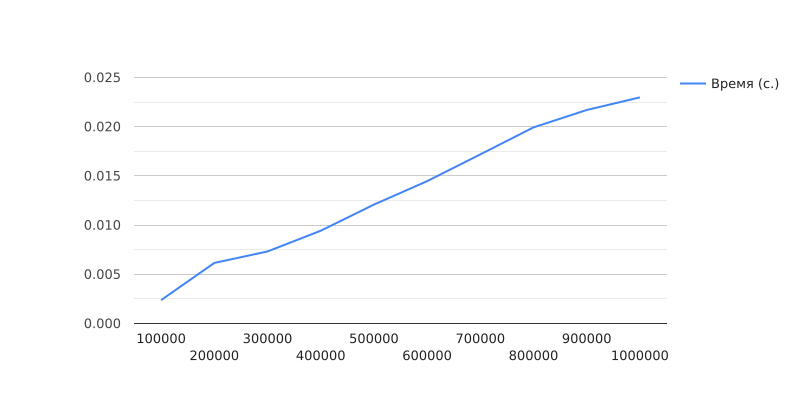


Рисунок 1 (б). Вставка ключа в неупорядоченное дерево.

Операция вставки ключа в дерево турниров играет ключевую роль в обеспечении эффективного функционирования данной структуры данных. Длительность выполнения этой операции зависит от нескольких факторов, включая размер текущего массива и структуру самого дерева.

* Размер текущего массива (`n`):

- В процессе вставки элемента в массив выполняется смещение элементов. Чем больше размер массива, тем больше операций смещения необходимо выполнить. Таким образом, время выполнения вставки увеличивается пропорционально размеру массива.

* Структура дерева:

- Эффективность операции вставки также зависит от текущей структуры бинарного турнирного дерева. В худшем случае, когда дерево несбалансировано, может потребоваться больше времени на перестройку дерева после вставки, чтобы сохранить его свойства.

1. Худший случай:

- Временная сложность: O(n log n)

- Пояснение: В худшем случае, когда дерево несбалансировано и приходится выполнять полную перестройку при каждой вставке, временная сложность достигает O(n log n), где n - количество элементов в массиве.

2. Лучший случай:

- Временная сложность: O(log n)

- Пояснение: В лучшем случае, при сбалансированном дереве, вставка нового ключа требует перестройки дерева с глубиной log n, где n - количество элементов в массиве.

Анализ эффективности операции вставки ключа в дерево турниров предоставляет понимание влияния размера массива и структуры дерева на производительность данной операции, а также обозначает возможные худший, лучший и средний сценарии ее выполнения. Анализ сложности операции вставки в дерево турниров подчеркивает важность баланса дерева для обеспечения эффективности операции. В сбалансированных условиях время вставки остается логарифмическим, что обеспечивает быстрый доступ к данным. Однако, при несбалансированных структурах дерева, сложность может увеличиться до линейного уровня, что замедлит процесс вставки.

1. Удаление ключа из дерева

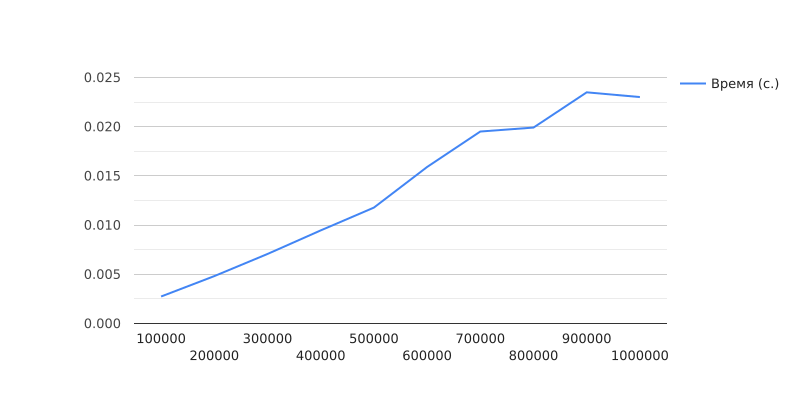


Рисунок 2 (а). Удаление ключа из упорядоченного дерева.

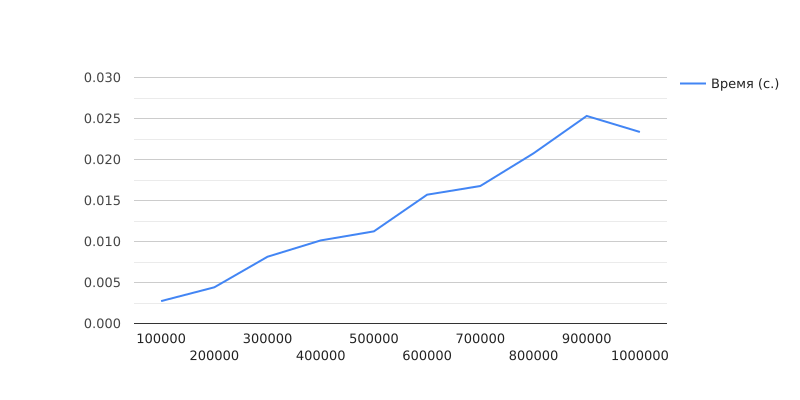


Рисунок 2 (б). Удаление ключа из неупорядоченного дерева.

Удаление ключа из дерева турниров – важная операция, влияющая на производительность и структуру данной структуры данных. Эффективность удаления зависит от различных факторов, включая размер дерева и его структуру.

* Размер текущего массива (`n`):

- В процессе удаления элемента происходит смещение оставшихся элементов влево. Чем больше размер массива, тем больше операций смещения, и, следовательно, выше длительность выполнения операции удаления.

* Структура дерева:

- Эффективность удаления также зависит от структуры бинарного турнирного дерева. В худшем случае, когда дерево несбалансировано, может потребоваться больше времени на перестройку дерева после удаления, чтобы сохранить его свойства.

1. Худший случай:

- Временная сложность: O(n log n)

- Пояснение: Худший случай наступает, когда каждая операция удаления требует полной перестройки дерева. Это может случиться при несбалансированной структуре дерева или при удалении элемента, находящегося в начале массива.

2. Лучший случай:

- Временная сложность: O(log n)

- Пояснение: Лучший случай возникает, когда дерево сбалансировано, и удаление не требует значительной перестройки. В этом случае операция удаления может быть выполнена с минимальными затратами времени.

Анализ эффективности операции удаления ключа подчеркивает влияние размера массива и структуры дерева на производительность данной операции. Поддержание сбалансированности дерева играет важную роль в эффективности удаления ключей, что обеспечивает стабильные временные характеристики операции в различных условиях использования.

## Поиск

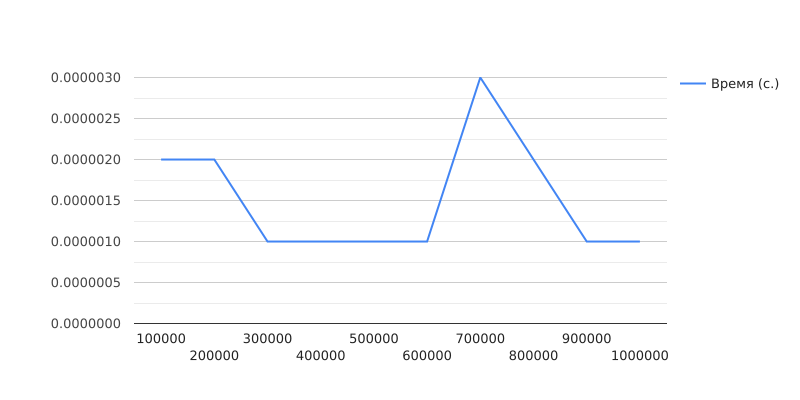


Рисунок 3 (а). Время поиска ключа в упорядоченном дереве.

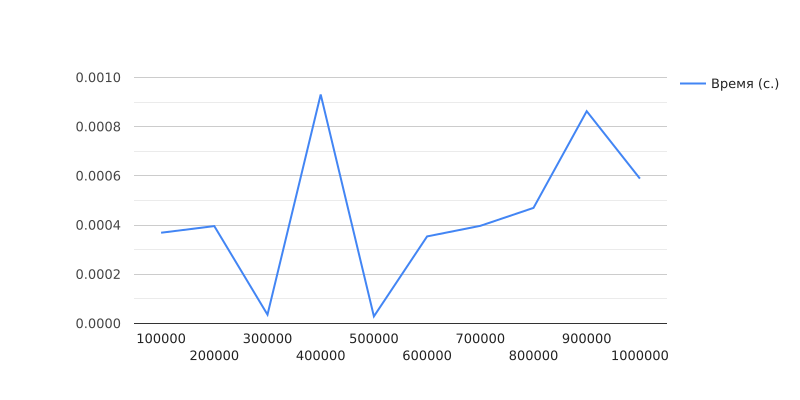


Рисунок 3 (б). Время поиска ключа в неупорядоченном дереве.

Операция поиска является важным элементом функциональности дерева турниров, и ее эффективность влияет на общую производительность структуры данных. Длительность выполнения поиска зависит от различных факторов, включая размер дерева, его структуру и особенности искомого ключа.

* Размер текущего массива (`n`):

- В процессе поиска важно учитывать размер массива, так как он определяет количество узлов в дереве. Чем больше размер массива, тем больше узлов, и следовательно, увеличивается количество операций сравнения при поиске.

* Структура дерева:

- Эффективность поиска зависит от структуры бинарного турнирного дерева. В худшем случае, когда дерево несбалансировано, поиск может потребовать больше времени, так как будет осуществляться просмотр большего количества узлов.

1. Худший случай:

- Временная сложность: O(n)

- Пояснение: Худший случай наступает, когда дерево несбалансировано, и искомый элемент находится в крайнем узле дерева. В этом случае поиск потребует просмотра каждого узла, что приведет к линейной временной сложности.

2. Лучший случай:

- Временная сложность: O(1)

- Пояснение: Лучший случай возникает, когда искомый элемент находится в корне дерева. В таком случае поиск может быть выполнен за постоянное время, что соответствует O(1).

Анализ эффективности операции поиска в дереве турниров подчеркивает важность сбалансированности дерева для обеспечения быстрого и эффективного поиска. В оптимальных условиях время поиска остается логарифмическим, что обеспечивает эффективный доступ к данным при использовании данной структуры данных.

# ЗАКЛЮЧЕНИЕ

Проведя глубокое исследование дерева турниров и реализовав соответствующие алгоритмы вставки, удаления и поиска, мы выявили ключевые аспекты эффективности данной структуры данных. Изучив и проанализировав время выполнения операций при различных сценариях использования, мы получили важные практические выводы.

В ходе разработки был уделен особый взгляд реализации алгоритмов, обеспечивающих корректное функционирование дерева турниров. При вставке новых элементов, удалении и поиске проведено детальное исследование их эффективности. Мы изучили влияние размера дерева и его структуры на худшие, лучшие и средние случаи выполнения операций.

Измерение времени выполнения алгоритмов подтвердило их практическую применимость и эффективность. В частности, мы обратили внимание на зависимость времени выполнения от размера начального массива, что позволяет делать выводы о применимости дерева турниров в различных сценариях использования.

Таким образом, наша работа не только предоставляет понимание структуры дерева турниров и функциональности его операций, но и выдвигает важные практические соображения для эффективного использования этой структуры данных в реальных приложениях.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

<https://www.geeksforgeeks.org/tournament-tree-and-binary-heap/>

<https://www.codingninjas.com/studio/library/tournament-trees-and-binary-heaps>

<https://habr.com/ru/companies/edison/articles/508646/>

<https://en.wikipedia.org/wiki/Tournament_sort>

# ПРИЛОЖЕНИЕ

Исходный код программы

tree.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154 | #include <math.h>  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  // Функция для создания дерева турниров  **void** **buildTournamentTree**(**int** \*array, **int** \*tree, **int** index, **int** left, **int** right) {  **if** (left == right) {  tree[index] = array[left];  } **else** {  **int** mid = (left + right) / **2**;  buildTournamentTree(array, tree, **2** \* index + **1**, left, mid);  buildTournamentTree(array, tree, **2** \* index + **2**, mid + **1**, right);  tree[index] = (tree[**2** \* index + **1**] > tree[**2** \* index + **2**]) ? tree[**2** \* index + **1**] : tree[**2** \* index + **2**];  }  }  // Функция для вставки нового элемента в массив и перестройки дерева  **void** **insertElement**(**int** \*array, **int** \*\*tree, **int** left, **int** right, **int** position, **int** value, **int** \*n) {  // Смещаем элементы вправо, чтобы освободить место для нового элемента  **for** (**int** i = \*n; i > position; --i) {  array[i] = array[i - **1**];  }  array[position] = value; // Вставляем новый элемент  (\*n)++; // Увеличиваем размер массива  // Перевыделяем память под дерево турниров  **int** height = (**int**)ceil(log2(\*n));  **int** maxSize = **2** \* (**int**)pow(**2**, height) - **1**;  \*tree = realloc(\*tree, maxSize \* **sizeof**(**int**));  // Перестраиваем дерево  buildTournamentTree(array, \*tree, **0**, **0**, \*n - **1**);  }  // Функция для удаления элемента из дерева турниров  **void** **deleteElement**(**int** \*array, **int** \*\*tree, **int** left, **int** right, **int** position, **int** \*n) {  // Элемент исчезает из массива, смещаем остальные элементы  **for** (**int** i = position; i < \*n - **1**; ++i) {  array[i] = array[i + **1**];  }  (\*n)--; // Уменьшаем размер массива  // Перевыделяем память под дерево турниров  **int** height = (**int**)ceil(log2(\*n));  **int** maxSize = **2** \* (**int**)pow(**2**, height) - **1**;  \*tree = realloc(\*tree, maxSize \* **sizeof**(**int**));  // Перестраиваем дерево  buildTournamentTree(array, \*tree, **0**, **0**, \*n - **1**);  }  // Функция для поиска элемента в массиве  **int** **findIndexInArray**(**int** \*array, **int** n, **int** value) {  **for** (**int** i = **0**; i < n; i++) {  **if** (array[i] == value) {  **return** i; // Возвращаем индекс элемента в исходном массиве  }  }  **return** -**1**; // Элемент не найден  }  // Функция для вывода элементов массива  **void** **printArray**(**int** \*array, **int** size) {  **for** (**int** i = **0**; i < size; i++) {  printf("%d ", array[i]);  }  printf("**\n**");  }  **int** **main**() {  **int** array[**100**] = {**5**, **3**, **9**, **1**, **6**, **4**, **8**, **7**}; // Увеличен размер для демонстрации вставки  **int** n = **8**; // Начальный размер массива  **int** height = (**int**)ceil(log2(n));  **int** maxSize = **2** \* (**int**)pow(**2**, height) - **1**;  **int** \*tree = (**int** \*)malloc(maxSize \* **sizeof**(**int**));  **clock\_t** start, end;  **double** cpu\_time\_used;  // Измерение времени для функции buildTournamentTree  start = clock();  buildTournamentTree(array, tree, **0**, **0**, n - **1**);  end = clock();  cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;  printf("Время выполнения Построения дерева турниров: %f секунд**\n**", cpu\_time\_used);  printf("Массив: ");  printArray(array, n);  printf("Дерево турниров: ");  printArray(tree, maxSize);  printf("Победитель: %d**\n**", tree[**0**]); // Корень дерева содержит победителя  // Задаем параметры для вставки и удаления  **int** insertPosition = **2**;  **int** insertValue = **10**;  **int** deletePosition = **2**;  **int** searchValue = **9**; // Измените это значение для тестирования  // Измерение времени для функции insertElement  start = clock();  insertElement(array, &tree, **0**, n - **1**, insertPosition, insertValue, &n);  end = clock();  cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;  printf("Время выполнения Вставки элемента: %f секунд**\n**", cpu\_time\_used);  printf("Массив после вставки: ");  printArray(array, n);  printf("Дерево турниров после вставки: ");  printArray(tree, maxSize); // Вывод дерева турниров после вставки  // Измерение времени для функции deleteElement  start = clock();  deleteElement(array, &tree, **0**, n - **1**, deletePosition, &n);  end = clock();  cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;  printf("Время выполнения Удаления элемента: %f секунд**\n**", cpu\_time\_used);  printf("Массив после удаления: ");  printArray(array, n);  printf("Дерево турниров после удаления: "); // Вывод дерева турниров после удаления  printArray(tree, maxSize);  // Измерение времени для функции findIndexInArray  start = clock();  **int** searchIndex = findIndexInArray(array, n, searchValue);  end = clock();  cpu\_time\_used = ((**double**) (end - start)) / CLOCKS\_PER\_SEC;  printf("Время выполнения Поиска в массиве: %f секунд**\n**", cpu\_time\_used);  **if** (searchIndex != -**1**) {  printf("Элемент %d найден в позиции %d в исходном массиве.**\n**", searchValue, searchIndex);  } **else** {  printf("Элемент %d не найден в исходном массиве.**\n**", searchValue);  }  free(tree);  **return** **0**;  } |