Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материала и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Дерево Фенвика

Выполнил студент гр. 3331506/20401 Хлусова К.А.

Преподаватель Ананьевский М. С.

Санкт-Петербург
2025

Оглавление

1. Введение

Данная курсовая работа посвящена изучению и практическому применению структуры данных, известной как дерево Фенвика, также называемой двоичным индексированным деревом (Binary Indexed Tree, BIT).

Дерево Фенвика представляет собой структуру данных, предназначенную для решения задач, связанных с вычислением префиксных сумм массива и выполнением операций по обновлению элементов массива.

Постановка задачи:

Зачастую в задачах программирования, например, в алгоритмах обработки массивов, возникает необходимость в быстром вычислении суммы элементов массива в заданном диапазоне (от индекса l до индекса r), а также необходимость оперативно обновлять значения отдельных элементов массива. Подход, подразумевающий простое суммирование элементов, обладает линейной временной сложностью O(n) для каждой операции запроса суммы, где n - размер массива. В случае, когда количество запросов на сумму и обновление на отрезке велико, такой подход становится неэффективным. Целью данной работы является изучение альтернативной структуры данных, позволяющей существенно ускорить выполнение этих операций.

Актуальность и значимость:

Дерево Фенвика предоставляет логарифмическую сложность (O(log n)) как для операции вычисления префиксной суммы, так и для операции обновления элемента. Эта особенность делает его незаменимым инструментом в задачах, где важна скорость обработки большого количества запросов на суммирование и обновление элементов массива. В сравнении с более сложными структурами данных, такими как деревья отрезков, дерево Фенвика отличается простотой реализации и меньшим объемом занимаемой памяти.

Область применения:

Дерево Фенвика широко применяется в различных областях, таких как:

- Обработка данных: Анализ данных, требующий быстрого вычисления агрегированных показателей на основе изменяющихся данных.

- Графика: Обработка изображений и видео, где необходимо быстро выполнять суммирование пиксельных значений в определенных областях.
- Финансовая аналитика: Анализ финансовых данных, требующий оперативного расчета сумм транзакций за определенный период времени.
- Игровое программирование: Разработка игр, где требуется быстрое обновление и анализ игрового мира.

2. Дерево Фенвика

2.1 Теоретические сведения

Алгоритм дерева Фенвика – это структура данных, дерево на массиве, которая обладает следующими свойствами:

- позволяет вычислять значение некоторой обратимой операции F на любом отрезке [L; R] за логарифмическое время;
- позволяет изменять значение любого элемента за O(log N);
- требует памяти O(N);

Операция F может представлять собой различные операции, но наиболее распространены вычисление суммы или произведения элементов на заданном интервале. В некоторых случаях, при наличии ограничений и определенных модификаций, F может быть операцией поиска максимального или минимального значения на интервале, а также другими операциями.

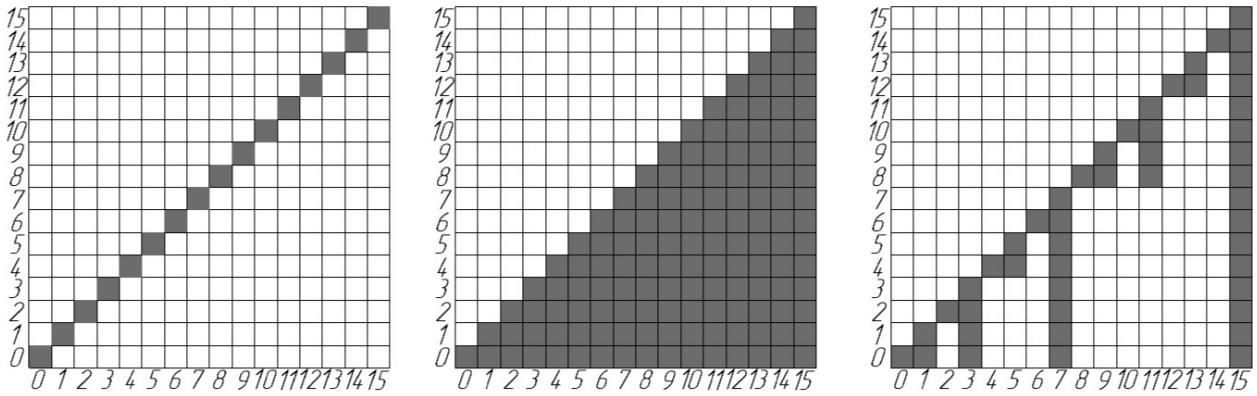

Рисунок 1 - Сравнение массивов и дерева Фенвика

2.2 Нахождение суммы

Для оценки времени, затрачиваемого для суммирования, было проведено измерение времени выполнения этих операций в зависимости от общего количества элементов суммирования. Результаты замеров показали следующую зависимость:

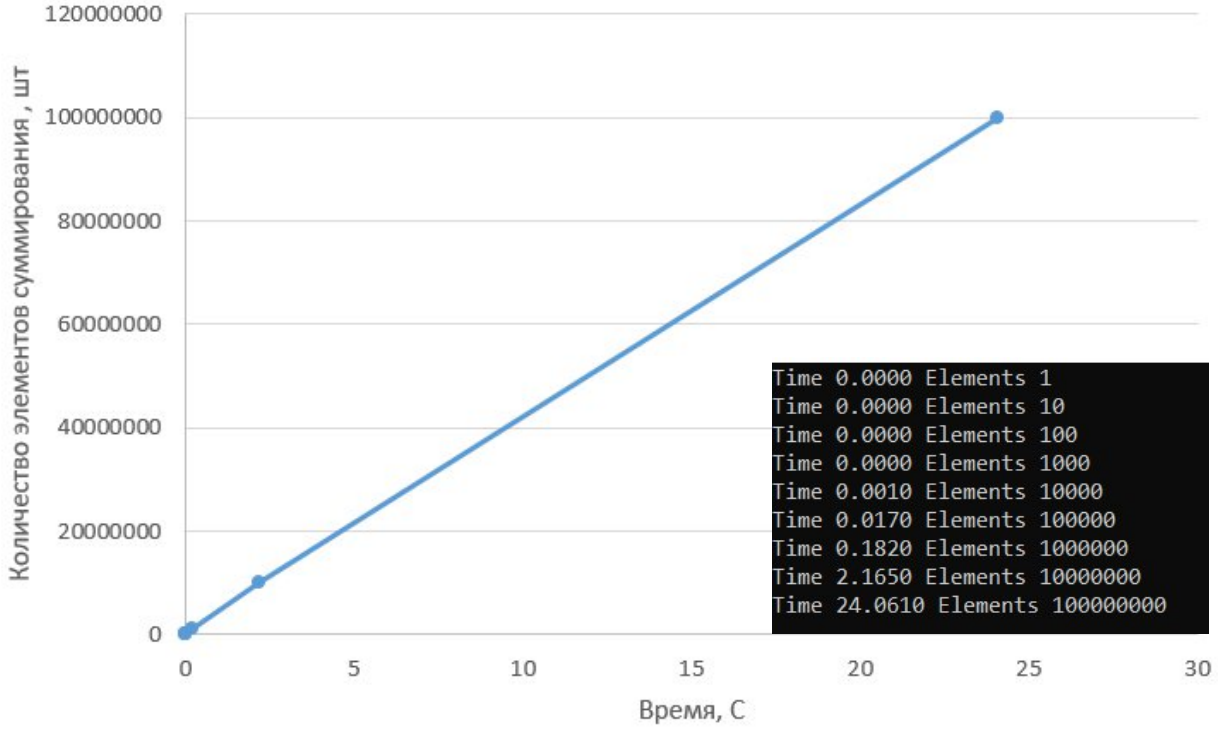

Рисунок 2 - График зависимости времени выполнения программы для суммы элементов от количества элементов суммирования

Для оценки времени работы алгоритма суммирования больших объёмов данных было проведено следующее исследование. В качестве исходных данных использовался массив, содержащий 1 миллиард (1 000 000 000) элементов. Суть эксперимента заключалась в измерении времени, затрачиваемого функцией на суммирование различных частей этого массива. Вначале был взят подмассив состоящий из одного элемента массива и было измерено время выполнения функции. Далее мы взяли подмассив состоящий из 10 элементов этого массива и также измерили время выполнения функции суммирования. Аналогичный образом, увеличивая каждый раз размер подмассива в 10 раз мы провели 9 измерений до тех пор, пока размер подмассива не стал равен размеру самого массива.

На основе этих данный был построен график. Для построения графика по горизонтальной оси отложено время вычисления суммы

подмассивов, а по вертикальной — размер подмассивов(количество суммированных элементов). График наглядно демонстрирует, как меняется время работы алгоритма при обработке всё большего объёма данных.

2.3 Нахождение минимума

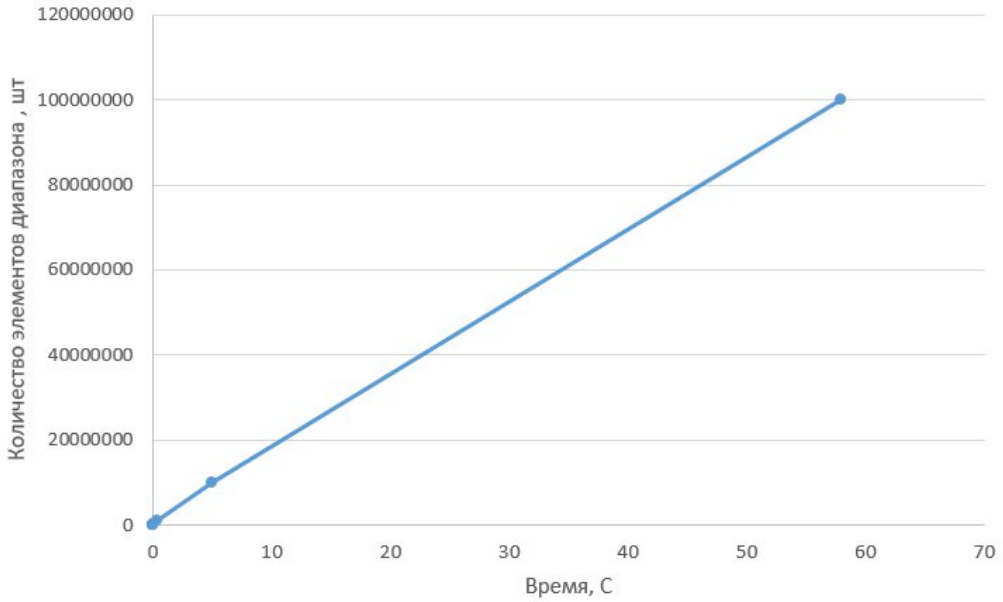

Рисунок 3 - График зависимости времени выполнения программы для поиска минимального элемента от количества элементов диапазона

Аналогичное исследование было проведено для определения времени, необходимого для поиска минимального элемента в подмассиве. Эксперимент заключался в следующем: измерялось время, затрачиваемое функцией на поиск минимума в подмассивах разной длины.

Мы начали с подмассива, состоящего из одного элемента, и зафиксировали время выполнения функции. Затем размер подмассива увеличили до 10 элементов и снова измерили время. Эту процедуру повторили, каждый раз увеличивая размер подмассива в 10 раз, пока он не достиг размера всего массива. Таким образом, было выполнено 9 измерений, отражающих зависимость времени поиска минимума от размера подмассива.

На основе этих данный был построен график. Для построения графика по горизонтальной оси отложено время поиска минимального элемента в подмассиве, а по вертикальной — размер подмассивов. Этот

график также наглядно демонстрирует, как меняется время работы алгоритма при обработке всё большего объёма данных.

2.4 Нахождение максимума

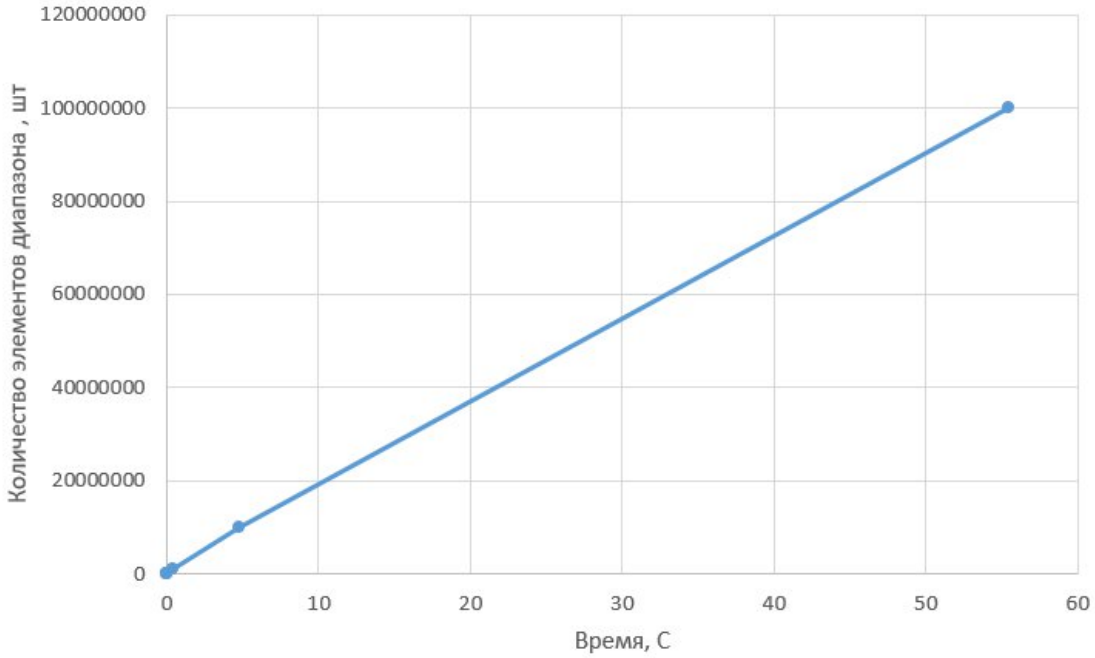

Рисунок 4 - График зависимости времени выполнения программы для поиска максимального элемента от количества элементов диапазона

Аналогичное исследование было проведено для определения времени, необходимого для поиска максимального элемента в подмассиве.

На основе этих данный был построен график. Для построения графика по горизонтальной оси отложено время поиска максимального элемента в подмассиве, а по вертикальной — размер подмассивов.

3. Заключение

В ходе работы было изучено и продемонстрировано применение дерева Фенвика, мощной и эффективной структуры данных для решения задач, связанных с вычислением префиксных сумм и обновлением элементов массивов. Было показано, что дерево Фенвика обеспечивает значительное повышение производительности по сравнению с наивными методами, достигая логарифмической временной сложности для операций запроса и обновления, что делает его незаменимым инструментом в задачах с большим объёмом данных и высокими требованиями к скорости обработки

4. Литература

1. Джордж Хайнеман, Гари Поллис, Стэнли Селков – Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017 г.
2. Рябко, Б.Я. Двоичные деревья, используемые для быстрого вычисления сумм. *Сибирский математический журнал,* 1989.
3. Кнут Д.Э. "Искусство программирования. Т. 4, А: Комбинаторные алгоритмы.2019
4. Скиена Стивен, Алгоритмы. Руководство по разработке, 2022
5. Генри С. Уоррен мл. – Алгоритмические трюки для программистов, 2014 г.

5. Приложение

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
std::vector<int> t;
int n;
void init(int nn) {
   n = nn;
   t.assign(n, 0);
}
int sum(int right) {
   int result = 0;
   for (; right >= 0; right = (right & (right + 1)) - 1) result += t[right];
   return result;
}
void inc(int idx, int delta) {
   for (; idx < n; idx = (idx | (idx + 1))) t[idx] += delta;
}
int sum(int left, int right) {
   return sum(right) - sum(left - 1);
}
void init(std::vector<int> a) {
   init((int)a.size());
   for (unsigned idx = 0; idx < a.size(); idx++) inc(idx, a[idx]);
}
int main() {
   unsigned long int k = 1000 * 1000 * 1000;
      clock_t start, end;
      init(k);
      for (unsigned long int i = 0; i < k; ++i) {
         inc(i, i);
      }
      for (int n = 1; n < 20; n += 1) {
         unsigned long int left = rand() % k;
         unsigned long int right = rand() % k;
         if (left > right) {
            std::swap(left, right);
         }
      start = clock();
      sum(left, right);
      end = clock();

      printf("Time %.7f ", ((double)end - start) / ((double)CLOCKS_PER_SEC));
      printf("Elements %d\n", left- right);
   }
}

#программа для минимума
/*
#include <iostream>
#include <algorithm>
```

```
#include <vector>

std:: vector<int> t;
int n;

const int INF = 1000 * 1000 * 1000;

void init(int nn) {
   n = nn;
   t.assign(n, INF);
}

int getmin(int right) {
   int result = INF;
   for (; right >= 0; right = (right & (right + 1)) - 1) result = std::min(result, t[right]);
   return result;
}

void update(int idx, int new_val) {
   for (; idx < n; idx = (idx | (idx + 1))) t[idx] = std::min(t[idx], new_val);
}

void init(std::vector<int> a) {
   init((int)a.size());
   for (unsigned idx = 0; idx < a.size(); idx++) update(idx, a[idx]);
}

int main() {
   for (int k = 1;k < 1000 * 1000 * 1000 * 100;k *= 10) {
      clock_t start, end;
      start = clock();
      init(k);
      for (int i = 0; i < k; i++) { update(i, i); }
      getmin(k - 1);
      end = clock();
      printf("Time %.4f ", ((double)end - start) / ((double)CLOCKS_PER_SEC));
      printf("Elements %d\n", k);
   }
}
*/

#программа для максимума
/*
#include <iostream>
#include <algorithm>
#include <vector>

std::vector<int> t;
int n;

const int INF = -1000 * 1000 * 1000;
```

```
void init(int nn) {
    n = nn;
    t.assign(n, INF);
}

int getmax(int right) {
    int result = INF;
    for (; right >= 0; right = (right & (right + 1)) - 1) result = std::max(result, t[right]);
    return result;
}

void update(int idx, int new_val) {
    for (; idx < n; idx = (idx | (idx + 1))) t[idx] = std::max(t[idx], new_val);
}

void init(std::vector<int> a) {
    init((int)a.size());
    for (unsigned idx = 0; idx < a.size(); idx++) update(idx, a[idx]);
}

int main() {
    for (int k = 1;k < 1000 * 1000 * 1000 * 100;k *= 10) {
        clock_t start, end;
        start = clock();
        init(k);
        for (int i = 0; i < k; i++) { update(i, i); }
        getmax(k - 1);
        end = clock();
        printf("Time %.4f ", ((double)end - start) / ((double)CLOCKS_PER_SEC));
        printf("Elements %d\n", k);
    }
*/
```