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

Fusion tree

| Выполнил студент | Пеалкиви Даниил Яковлевич |
| --- | --- |
|  | Ф.И.О. |

| Группы | ИC-242 |
| --- | --- |
|  |  |

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

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

Новосибирск – 2023

**Оглавление**

ВВЕДЕНИЕ 3

ОПИСАНИЕ СТРУКТУРЫ 4

1. Вставка ключа в дерево 4
2. Удаление ключа из дерева 7
3. Поиск ключа в дереве 9
4. Разбиение узла в дереве 10
5. Обход узлов дерева 12

АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ 13

1. Вставка ключа в дерево 13
2. Удаление ключа из дерева 14
3. Поиск ключа в дереве 15

ЗАКЛЮЧЕНИЕ 16

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 17

ПРИЛОЖЕНИЕ 18

# Введение

С каждым днем объемы данных, которые необходимо эффективно обрабатывать, стремительно увеличиваются. В этом контексте деревья поиска играют важную роль в обеспечении быстрого и эффективного доступа к данным. Однако традиционные структуры данных, такие как B-деревья или AVL-деревья, могут столкнуться с ограничениями в условиях постоянного роста данных.

В рамках данной курсовой работы мною была изучена и реализована такая структура данных как дерево слияния. Дерево слияния представляет собой инновационный метод структурирования данных, который стремится преодолеть эти ограничения и обеспечить высокую производительность поиска в условиях динамично изменяющихся наборов данных. Это статическая структура данных, которая была впервые предложена в 1990 году М. Фредманом (M. Fredman) и Д. Уиллардом (D. Willard).

Дерево слияния предоставляет ряд полезных применений в области обработки данных и поиска:

1. Базы данных и поиск: Fusion Tree может использоваться в базах данных для эффективного выполнения операций поиска. Его уникальные характеристики позволяют обрабатывать запросы на поиск с высокой скоростью, что особенно важно в системах управления большими объемами данных.
2. Системы управления файлами: в системах управления файлами Fusion Tree может быть применен для быстрого поиска и доступа к информации в больших файлах или базах данных.
3. Обработка потоков данных: в сценариях обработки потоков данных Fusion Tree может использоваться для эффективной фильтрации и поиска в поступающих потоках данных.

# Описание структуры

Fusion Tree - это структура данных, представляющая собой B-дерево, но с дополнительными особенностями, которые позволяют ему эффективно поддерживать операции поиска, вставки и удаления элементов. Основной идеей Fusion Tree является объединение двух концепций: B-дерева и сортированных массивов.

1. Вставка ключа в дерево

| **Алгоритм 1** insert | |
| --- | --- |
| 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  383940414243444546474849505152535455565758596061626364656667686970 | void insert(int a)  {  int i, temp;  x = root;  if (x == NULL)  {  root = init();  x = root;  }  else  {  if (x->leaf == true && x->n == 5)  {  temp = split\_child(x, -1);  x = root;  for (i = 0; i < (x->n); i++)  {  if ((a > x->data[i]) && (a < x->data[i + 1]))  {  i++;  break;  }  else if (a < x->data[0])  {  break;  }  else  {  continue;  }  }  x = x->child\_ptr[i];  }  else  {  while (x->leaf == false)  {  for (i = 0; i < (x->n); i++)  {  if ((a > x->data[i]) && (a < x->data[i + 1]))  {  i++;  break;  }  else if (a < x->data[0])  {  break;  }  else  {  continue;  }  }  if ((x->child\_ptr[i])->n == 5)  {  temp = split\_child(x, i);  x->data[x->n] = temp;  x->n++;  continue;  }  else  {  x = x->child\_ptr[i];  }  }  }  }  x->data[x->n] = a;  sort(x->data, x->n);  x->n++;  } |

Вставка начинается с корня дерева. Если дерево пусто, создается новый корень. После этого производится поиск листового узла, в который должен быть вставлен ключ. Это делается путем рекурсивного спуска по дереву от корня к листовым узлам. Если текущий узел является листом, то он выбирается для вставки. Когда найден листовой узел для вставки, ключ вставляется в его упорядоченный массив.

Если листовой узел оказывается переполненным после вставки, происходит его разделение. Если при вставке ключа узел становится переполненным, то происходит его разделение. Средний элемент массива перемещается в родительский узел, и узел разделяется на два. В случае, если у соседних узлов недостаточно элементов, они объединяются. Если разделение произошло в корне, создается новый корень, и два разделенных узла становятся его дочерними.

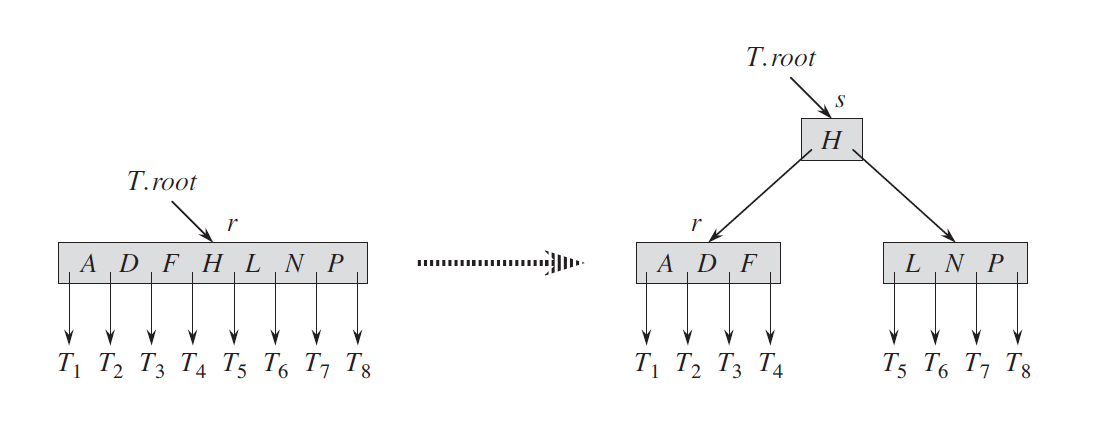


Рисунок 1. Вставка ключа в дерево

1. Удаление ключа из дерева

| **Алгоритм 2** remove | |
| --- | --- |
| 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 | void remove(int key, FusionTreeNode\* node)  {  if (node == NULL)  return;  int i = 0;  while (i < node->n && key > node->data[i])  i++;  if (i < node->n && key == node->data[i])  {  for (int j = i; j < node->n - 1; j++)  node->data[j] = node->data[j + 1];  node->n--;  return;  }  if (node->leaf)  {  return;  }  bool lastChild = (i == node->n);  remove(key, node->child\_ptr[i]);  if (lastChild && i > 0 && node->child\_ptr[i]->n < 2)  {  FusionTreeNode\* child = node->child\_ptr[i];  FusionTreeNode\* sibling = node->child\_ptr[i - 1];  child->data[0] = node->data[i - 1];  child->n++;  node->data[i - 1] = sibling->data[sibling->n - 1];  sibling->n--;  child->child\_ptr[child->n] = sibling->child\_ptr[sibling->n + 1];  sibling->child\_ptr[sibling->n + 1] = NULL;  }  } |

Начнем с корня и двигаемся вниз по дереву, чтобы найти узел, содержащий удаляемый ключ. Если ключ находится в листе, просто удалим его. Если ключ находится во внутреннем узле, продолжим поиск в поддереве, связанном с соответствующим потомком. Если узел, содержащий удаляемый ключ, является листом, просто удалим ключ из узла. После этого нужно проверить, не стал ли узел "слишком маленьким" (то есть содержит менее двух ключей).

Если да, возможно, потребуется объединение соседних узлов или перемещение ключей. Если узел, содержащий удаляемый ключ, является внутренним узлом, найдем предшественника или преемника ключа (или узла) для замены удаляемого ключа. Затем рекурсивно удалим предшественника или преемника из соответствующего поддерева. После удаления ключа необходимо обновить структуру дерева, чтобы удовлетворять его инвариантам. Это может включать в себя перемещение ключей между узлами, объединение узлов или разделение узлов.

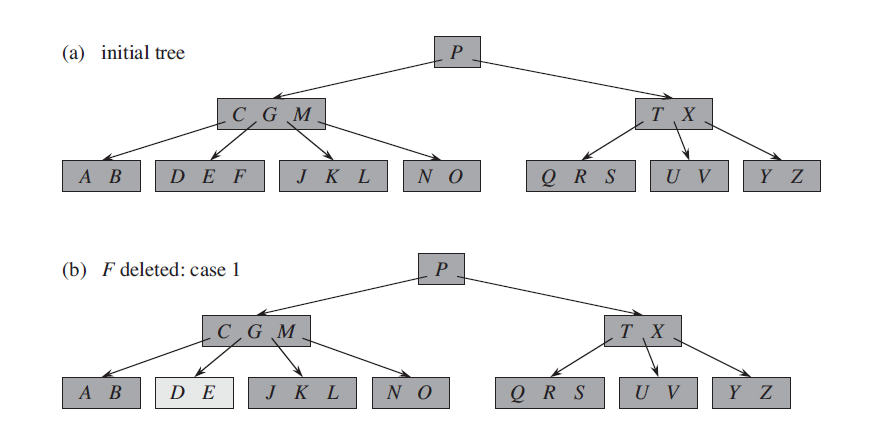


Рисунок 2. Удаление ключа из дерева

1. Поиск ключа в дереве

| **Алгоритм 3** search | |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | FusionTreeNode\* search(FusionTreeNode\* node, int value) {  if (node == NULL) {  return NULL;  }  int i = 0;  while (i < node->n && value > node->data[i]) {  i++;  }  if (i < node->n && value == node->data[i]) {  return node; // Найдено значение в текущем узле  }  if (node->leaf) {  return NULL; // Не найдено значение в листе  }  return search(node->child\_ptr[i], value); // Рекурсивный поиск в поддереве  } |

Функция поиска ключа в дереве просматривает дерево, начиная с корня и двигаясь вниз по дереву в соответствии с порядком ключей в узлах. Начиная с корня, она сравнивает ключ, который вы ищете, с ключами в текущем узле. Если ключ меньше текущего ключа, переходит к левому поддереву. Если ключ больше текущего ключа, переходит к правому поддереву. Если ключ совпадает с текущим ключом, возвращает этот узел, так как ключ найден. Если достигнут лист (узел без детей), и ключ не найден, возвращает NULL, указывая, что ключ отсутствует в дереве. Поиск выполняется рекурсивно, что означает, что для поиска в поддереве он вызывает себя с текущим поддеревом в качестве нового корня. Этот процесс продолжается, пока ключ не будет найден или не будет достигнут лист.

1. Разбиение узла в дереве

| **Алгоритм 4** split\_child | |
| --- | --- |
| 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  404142434445464748 | int split\_child(FusionTreeNode\* x, int i)  {  int j, mid;  FusionTreeNode\* np1, \* np3, \* y;  np3 = init();  np3->leaf = true;  if (i == -1)  {  mid = x->data[2];  x->data[2] = 0;  x->n--;  np1 = init();  np1->leaf = false;  x->leaf = true;  for (j = 3; j < 5; j++)  {  np3->data[j - 3] = x->data[j];  np3->child\_ptr[j - 3] = x->child\_ptr[j];  np3->n++;  x->data[j] = 0;  x->n--;  }  for (j = 0; j < 6; j++)  {  x->child\_ptr[j] = NULL;  }  np1->data[0] = mid;  np1->child\_ptr[np1->n] = x;  np1->child\_ptr[np1->n + 1] = np3;  np1->n++;  root = np1;  }  else  {  y = x->child\_ptr[i];  mid = y->data[2];  y->data[2] = 0;  y->n--;  for (j = 3; j < 5; j++)  {  np3->data[j - 3] = y->data[j];  np3->n++;  y->data[j] = 0;  y->n--;  }  x->child\_ptr[i + 1] = y;  x->child\_ptr[i + 1] = np3;  }  return mid;  } |

Функция разбиения узла в дереве используется для вставки нового ключа в узел, который уже содержит максимальное количество ключей. Разбиение происходит в следующих случаях, если узел является корнем, и он полон, то создается новый корень. Существующий корень становится его дочерним элементом. Если узел не является корнем, и он полон, то он разбивается на два узла. Средний ключ перемещается вверх, и родительскому узлу добавляется новый ключ и указатель на новый узел. Если узел является листом, то разбиение происходит просто, и новый ключ вставляется в узел. В случае фьюжн-дерева разбиение осуществляется на три части. Средний ключ остается в текущем узле, а левая и правая части используются для создания двух новых узлов, которые будут дочерними для текущего узла. Это обеспечивает балансировку дерева и позволяет эффективно поддерживать упорядоченное множество данных.

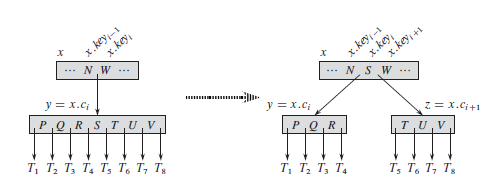


Рисунок 3. Разбиение узла в дереве

1. Обхода узлов дерева

| **Алгоритм 5** traverse | |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | void traverse(FusionTreeNode\* p)  {  cout << endl;  int i;  for (i = 0; i < p->n; i++)  {  if (p->leaf == false)  {  traverse(p->child\_ptr[i]);  }  cout << " " << p->data[i];  }  if (p->leaf == false)  {  traverse(p->child\_ptr[i]);  }  cout << endl;  } |

Функция traverse предназначена для обхода фьюжн-дерева и вывода его содержимого в консоль. Она рекурсивно обходит узлы дерева, начиная с корня, и выводит ключи, содержащиеся в узлах. В каждом узле выводятся ключи, а затем происходит рекурсивный вызов traverse для каждого дочернего узла.

Процесс обхода и вывода выглядит следующим образом:

1. Посещение текущего узла.
2. Вывод всех ключей, содержащихся в текущем узле.
3. Рекурсивный вызов traverse для каждого дочернего узла.

Таким образом, функция traverse позволяет визуализировать структуру фьюжн-дерева и увидеть упорядоченный набор ключей в порядке их следования в дереве.

# Анализ эффективности алгоритмов

В данной главе будет проводится асимптотический анализ описанных выше алгоритмов вместе с их экспериментальным исследованием.

1. Вставка ключа в дерево

| Вставка ключа в дерево | |
| --- | --- |
|
| Кол-во элементов | Время выполнения |
| 10000 | 0.000011 |
| 20000 | 0.000014 |
| 30000 | 0.000017 |
| 40000 | 0.000018 |
| 50000 | 0.000021 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |



Рисунок 4. Время вставки ключа

1. Удаление ключа из дерева

| Удаление ключа из дерева | |
| --- | --- |
|
| Кол-во элементов | Время выполнения |
| 10000 | 0.000017 |
| 20000 | 0.000036 |
| 30000 | 0.000052 |
| 40000 | 0.000070 |
| 50000 | 0.000087 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

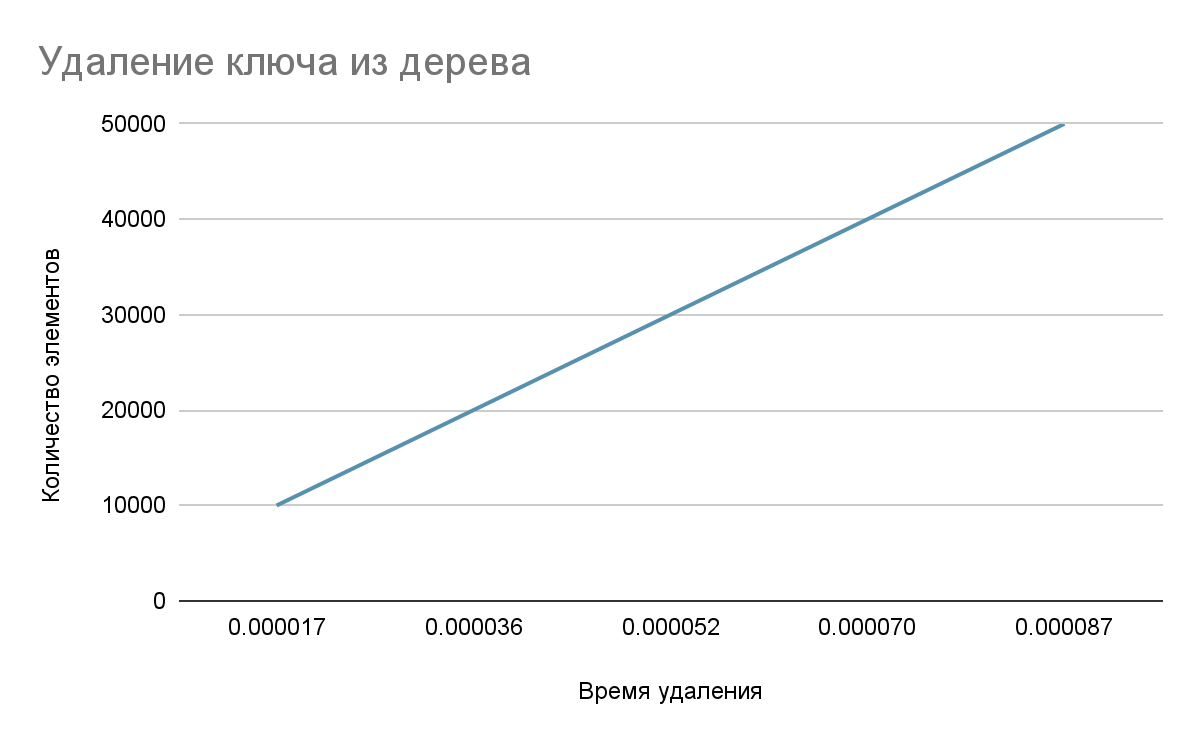
****

Рисунок 5. Время удаление ключа из дерева

1. Поиск ключа в дереве

| Поиск ключа в дереве | |
| --- | --- |
|
| Кол-во элементов | Время выполнения |
| 10000 | 0.000137 |
| 20000 | 0.000192 |
| 30000 | 0.000201 |
| 40000 | 0.000266 |
| 50000 | 0.000340 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

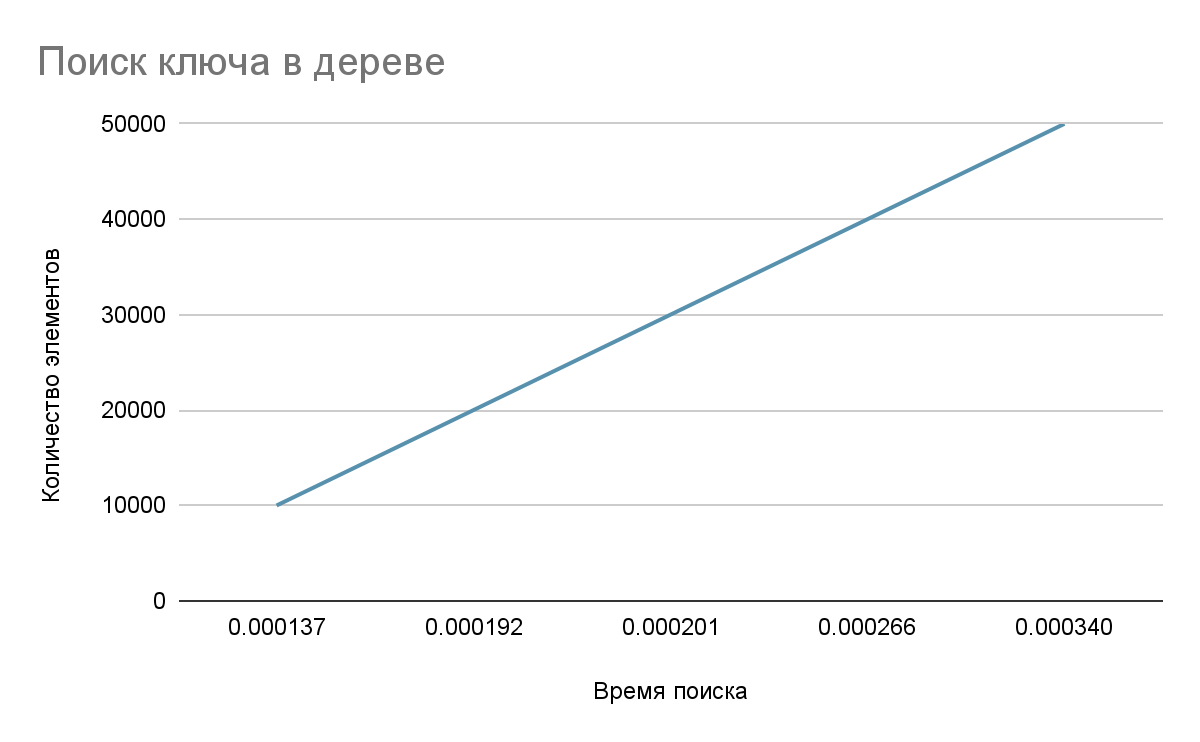


Рисунок 6. Время поиска ключа в дереве

# 

# Заключение

Исследование и реализация Fusion Tree позволяют нам оценить его эффективность в управлении упорядоченными данными. Структура обеспечивает эффективные операции вставки, поиска и удаления, что делает ее привлекательной для определенных приложений.

Балансировка и упорядоченность: Fusion Tree обеспечивает автоматическую балансировку при вставке и удалении элементов, что поддерживает упорядоченность данных и предотвращает деградацию производительности.

Применение в различных областях: структура данных, подобная Fusion Tree, может быть полезной в сценариях, где требуется эффективная работа с большими объемами упорядоченных данных, таких как базы данных, системы поиска, управления версиями и другие.

Сложность операций: в дереве слияния, в лучшем и среднем случае, операции вставки и удаления будут иметь асимптотическую сложность O(log N), где n - количество элементов в дереве. Это связано с тем, что дерево слияния обеспечивает балансировку при вставке и удалении элементов, что позволяет поддерживать логарифмическую высоту дерева.

Однако стоит отметить, что в худшем случае, операции вставки и удаления могут иметь сложность O(n), если дерево неудачно сбалансировано. В таких случаях, эффективность дерева слияния может снижаться, и возможно, потребуется дополнительная балансировка для поддержания оптимальной структуры.

В целом, Fusion Tree представляет собой мощную структуру данных, способную обеспечить эффективное управление упорядоченными данными, и его использование может быть оправданным в зависимости от конкретных требований вашего приложения.

# Список использованной литературы

1. IFMO. Fusion tree. URL: <https://neerc.ifmo.ru/wiki/index.php?title=Fusion_tree>
2. Wikipedia. Fusion tree. URL: <https://en.wikipedia.org/wiki/Fusion_tree>
3. IFMO. B-tree. URL: [https://neerc.ifmo.ru/wiki/index.php?title=B-дерево](https://neerc.ifmo.ru/wiki/index.php?title=B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)

# Приложение

| 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 | #include<stdio.h>  #include<conio.h>  #include<iostream>  #include <chrono>  #include <vector>  using namespace std;  int k = 0;  struct FusionTreeNode  {  int\* data;  FusionTreeNode\*\* child\_ptr;  bool leaf;  int n;  }\*root = NULL, \* np = NULL, \* x = NULL;  FusionTreeNode\* init()  {  int i;  np = new FusionTreeNode;  np->data = new int[5];  np->child\_ptr = new FusionTreeNode \* [6];  np->leaf = true;  np->n = 0;  for (i = 0; i < 6; i++)  {  np->child\_ptr[i] = NULL;  }  return np;  }  FusionTreeNode\* search(FusionTreeNode\* node, int value) {  if (node == NULL) {  return NULL;  }  int i = 0;  while (i < node->n && value > node->data[i]) {  i++;  }  if (i < node->n && value == node->data[i]) {  return node; // Найдено значение в текущем узле  }  if (node->leaf) {  return NULL; // Не найдено значение в листе  }  return search(node->child\_ptr[i], value); // Рекурсивный поиск в поддереве  }  void traverse(FusionTreeNode\* p)  {  cout << endl;  int i;  for (i = 0; i < p->n; i++)  {  if (p->leaf == false)  {  traverse(p->child\_ptr[i]);  }  cout << " " << p->data[i];  }  if (p->leaf == false)  {  traverse(p->child\_ptr[i]);  }  cout << endl;  }  void sort(int\* p, int n)  {  int i, j, temp;  for (i = 0; i < n; i++)  {  for (j = i; j <= n; j++)  {  if (p[i] > p[j])  {  temp = p[i];  p[i] = p[j];  p[j] = temp;  }  }  }  }  int split\_child(FusionTreeNode\* x, int i)  {  int j, mid;  FusionTreeNode\* np1, \* np3, \* y;  np3 = init();  np3->leaf = true;  if (i == -1)  {  mid = x->data[2];  x->data[2] = 0;  x->n--;  np1 = init();  np1->leaf = false;  x->leaf = true;  for (j = 3; j < 5; j++)  {  np3->data[j - 3] = x->data[j];  np3->child\_ptr[j - 3] = x->child\_ptr[j];  np3->n++;  x->data[j] = 0;  x->n--;  }  for (j = 0; j < 6; j++)  {  x->child\_ptr[j] = NULL;  }  np1->data[0] = mid;  np1->child\_ptr[np1->n] = x;  np1->child\_ptr[np1->n + 1] = np3;  np1->n++;  root = np1;  }  else  {  y = x->child\_ptr[i];  mid = y->data[2];  y->data[2] = 0;  y->n--;  for (j = 3; j < 5; j++)  {  np3->data[j - 3] = y->data[j];  np3->n++;  y->data[j] = 0;  y->n--;  }  x->child\_ptr[i + 1] = y;  x->child\_ptr[i + 1] = np3;  }  return mid;  }  void insert(int a)  {  int i, temp;  x = root;  if (x == NULL)  {  root = init();  x = root;  }  else  {  if (x->leaf == true && x->n == 5)  {  temp = split\_child(x, -1);  x = root;  for (i = 0; i < (x->n); i++)  {  if ((a > x->data[i]) && (a < x->data[i + 1]))  {  i++;  break;  }  else if (a < x->data[0])  {  break;  }  else  {  continue;  }  }  x = x->child\_ptr[i];  }  else  {  while (x->leaf == false)  {  for (i = 0; i < (x->n); i++)  {  if ((a > x->data[i]) && (a < x->data[i + 1]))  {  i++;  break;  }  else if (a < x->data[0])  {  break;  }  else  {  continue;  }  }  if ((x->child\_ptr[i])->n == 5)  {  temp = split\_child(x, i);  x->data[x->n] = temp;  x->n++;  continue;  }  else  {  x = x->child\_ptr[i];  }  }  }  }  x->data[x->n] = a;  sort(x->data, x->n);  x->n++;  }  void remove(int key, FusionTreeNode\* node)  {  if (node == NULL)  return;  int i = 0;  while (i < node->n && key > node->data[i])  i++;  if (i < node->n && key == node->data[i])  {  for (int j = i; j < node->n - 1; j++)  node->data[j] = node->data[j + 1];  node->n--;  return;  }  if (node->leaf)  {  return;  }  bool lastChild = (i == node->n);  remove(key, node->child\_ptr[i]);  if (lastChild && i > 0 && node->child\_ptr[i]->n < 2)  {  FusionTreeNode\* child = node->child\_ptr[i];  FusionTreeNode\* sibling = node->child\_ptr[i - 1];  child->data[0] = node->data[i - 1];  child->n++;  node->data[i - 1] = sibling->data[sibling->n - 1];  sibling->n--;  child->child\_ptr[child->n] = sibling->child\_ptr[sibling->n + 1];  sibling->child\_ptr[sibling->n + 1] = NULL;  }  }  int main()  {  int i, n, t;  cout << "Enter the number of elements to be inserted: \n";  cin >> n;  for (i = 0; i < n; i++)  {  cout << "Enter the binary element: \n";  cin >> t;  insert(t);  }  cout << "Traversal of constructed fusion tree: \n";  traverse(root);  int valueToSearch = 101;  FusionTreeNode\* result = search(root, valueToSearch);  if (result != NULL) {  cout << "Value " << valueToSearch << " find\n" << endl;  }  else {  cout << "Value " << valueToSearch << " not find\n" << endl;  }  int key\_to\_remove;  cout << "Enter the key to remove: \n";  cin >> key\_to\_remove;  remove(key\_to\_remove, root);  cout << "Traversal after remove: \n";  traverse(root);  } |
| --- | --- |