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

ТЕРНАРНОЕ ДЕРЕВО ПОИСКА

|  |  |
| --- | --- |
| Выполнил студент | Кулик П.Е. |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИС241 |
|  |  |

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

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

Новосибирск – 2023

Оглавление

[ВВЕДЕНИЕ 4](#_Toc151597683)

[ОПИСАНИЕ СТРУКТУРЫ 5](#_Toc151597684)

[ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ 7](#_Toc151597685)

[1. Вставка ключа в дерево 7](#_Toc151597686)

[2. Удаление ключа из дерева 12](#_Toc151597687)

[2.1 Вспомогательные операции для удаления 16](#_Toc151597688)

[2.1.1 start\_of\_key 16](#_Toc151597689)

[2.1.2 end\_of\_key 17](#_Toc151597690)

[2.1.3 free\_key 19](#_Toc151597691)

[2.1.4 tst\_restore\_prop\_any\_case 20](#_Toc151597692)

[2.1.5 tst\_restore\_prop 21](#_Toc151597693)

[3. Удаление всего дерева 21](#_Toc151597694)

[4. Вывод списка слов 22](#_Toc151597695)

[4.1 tst\_print\_all\_words 22](#_Toc151597696)

[4.2 tst\_print\_words\_from\_node 23](#_Toc151597697)

[5. Поиск 23](#_Toc151597698)

[5.1 tst\_lookup 24](#_Toc151597699)

[5.2 tst\_find\_sym 25](#_Toc151597700)

[6. Префиксный поиск 25](#_Toc151597701)

[6.1 tst\_prefix\_search 25](#_Toc151597702)

[АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ 27](#_Toc151597703)

[1. Поиск ключа 27](#_Toc151597704)

[2. Добавление ключей в дерево 29](#_Toc151597705)

[3. Удаление ключа из дерева 31](#_Toc151597706)

[ЗАКЛЮЧЕНИЕ 33](#_Toc151597707)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 34](#_Toc151597708)

[ПРИЛОЖЕНИЕ 35](#_Toc151597709)

# ВВЕДЕНИЕ

На сегодняшний день разработано и исследовано множество различных структур данных, каждая из которых обладает своими особенностями и может эффективно применяться в подходящих задачах.

В рамках данной курсовой работы мною была изучена и реализована такая структура данных как тернарное дерево поиска. Впервые эта структура была описана в статье под названием "A tree for associative searching" («Дерево для ассоциативного поиска»), написанной Найджелом П. Калдером (Nigel P. Chapman) и Эрнестом С. Коузенсом (Ernest S. Couden). Эта статья была опубликована в журнале "Communications of the ACM" в июне 1971 года (Volume 14, Issue 6, Pages 427-432).

Исследуемая структура может обладать множеством различных полезных применений:

1. Автозавершение и поиск по словарям: TST может быть использовано для реализации функциональности автозавершения в текстовых редакторах, поиске по словарям или поиске слов в больших наборах данных.
2. Индексация текста: тернарные деревья поиска могут быть применены для построения индекса текста, что ускоряет операции поиска подстрок в текстовых данных.
3. Словари и поиск слов: тернарные деревья поиска могут быть применены для хранения словарей и эффективного поиска слов в них.
4. Реализация ассоциативных массивов: TST может служить основой для реализации структур данных, подобных ассоциативным массивам, где ключами являются строки.

# ОПИСАНИЕ СТРУКТУРЫ

Основная суть тернарного дерева поиска заключается в том, что каждый узел хранит в себе один символ ключа, представляющего собой строку и может иметь до трёх дочерних узлов, а именно нижний (low), равный (equal) и верхний (high). Нижний и верхний узлы используются по тому же принципу, что левый и правый в бинарном дереве поиска, то есть в нижний дочерний узел попадает символ, который располагается ближе к началу алфавита, чем его родитель, а в верхний дочерний узел попадает символ, который находится дальше от начала алфавита, чем его родитель. Равный узел служит для того, чтобы хранить в себе те символы, которые являются продолжением хранящегося в дереве ключа. Помимо символа ключа и указателей на дочерние узлы в каждом узле имеется также метка, хранящая в себе одно из двух возможных состояний узла, которая нужна для того, чтобы проверить, является ли символ, хранящийся в узле, завершающим символом ключа или нет.

На рисунке 1 представлен пример дерева, содержащего следующие ключи: NEVER, NIGHT, LOW, LOWER, LOUNGE, RUST, ROOM, ROOSTER.

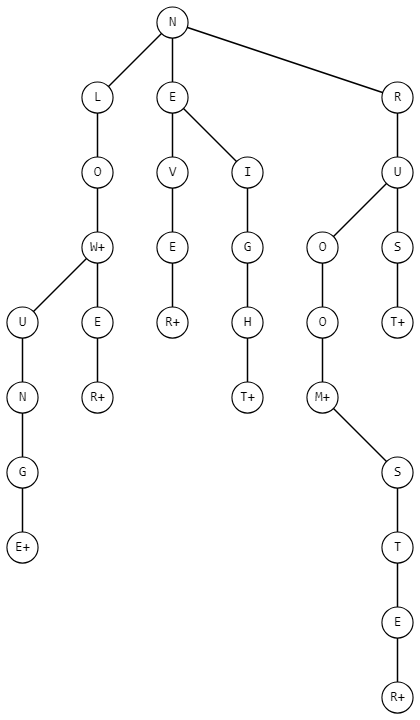


Рисунок 1. Пример тернарного дерева.

# ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ

## Вставка ключа в дерево

|  |  |
| --- | --- |
| Алгоритм 1 tst\_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  38 | function tst\_insert(tree, key):  node = tree  prev = NULL  indicator = NONE  while node is not NULL:  if key[0] < node.ch:  indicator = LO  prev = node  node = node.lokid  else if key[0] > node.ch:  indicator = HI  prev = node  node = node.hikid  else:  indicator = EQ  prev = node  node = node.eqkid  key = key + 1  if key[0] is NULL:  prev.end = true  return tree  node = create\_node(key[0])  if tree is NULL:  tree = node  switch indicator:  case NONE: break  case LO: prev.lokid = node  case HI: prev.hikid = node   case EQ: prev.eqkid = node   prev = node  key = key + 1  while key[0] is not NULL:  node = create\_node(key[0])  key = key + 1  prev.eqkid = node  prev = node  node.end = true  return tree |

Для того, чтобы вставить в дерево новый ключ, требуется сначала найти то место, с которого следует начать вставку. Если дерево пустое, то новый ключ становится корнем дерева, а вот если в дереве уже были ключи, то начинается поиск узла node, с которого начнётся добавление нового ключа.

Изначально узлу node присваивается значение корня дерева. Во время поиска места под новый ключ, первый символ ключа сравнивается с теми символами, которые уже хранятся в дереве. Если символ нового ключа меньше (находится ближе к началу алфавита), чем тот ключ, который уже есть, то узел node заменяется на его нижний дочерний узел, а если символ больше (дальше от начала алфавита), чем тот, что уже добавлен, то узел node заменяется на верхний дочерний узел. При каждой замене узла node в переменную prev сохраняется его предыдущее состояние, а в переменную indicator сохраняется информация о том, на какой из дочерних узлов был заменён узел node для того, чтобы в дальнейшем можно было связать новые узлы с теми, что уже были добавлены. Если возникает такая ситуация, что первый символ ключа уже был добавлен в дерево, то узел node заменяется на свой равный дочерний узел, а исследуемый символ ключа заменяется на его следующий символ, после чего поиск продолжается дальше.

Если в ходе поиска будут найдены все символы ключа, то узел prev, в котором в таком случае хранится последний символ ключа, будет помечен как узел с завершающим символом вне зависимости от того, какая у него до этого была метка, и алгоритм завершит свою работу. Если до запуска алгоритма узел prev был помечен как узел без ключа, то теперь в дереве будет новый ключ, для которого уже были готовы все узлы, а если метка окончания ключа уже стояла, то повторная метка никак не повлияет на дерево.

После того, как место для новых узлов найдено, начинается добавление новых узлов в дерево. При этом первый новый узел может быть по отношению к его родителю как равным, так и нижним или верхним, а вот все следующие за ним узлы будут равными дочерними узлами.

Особенность алгоритма вставки ключей в тернарное дерево поиска заключается в том, что если в дереве уже хранится часть нового ключа, то те символы, которых не было в дереве, будут добавлены в нижнее или верхнее поддерево первого узла с несовпадающим символом, следующего за последним узлом с совпадающим символом. Таким образом, если узел является нижним или верхним по отношению к своему родителю, то символ, хранящийся в родительском узле, точно не является частью ключа, часть которого хранится в дочернем узле, но при этом другая часть ключа может хранится ближе к корню.

На рисунке 2 можно наглядно увидеть работу алгоритма добавления ключей в дерево на примере добавления ключа LOUNGE.

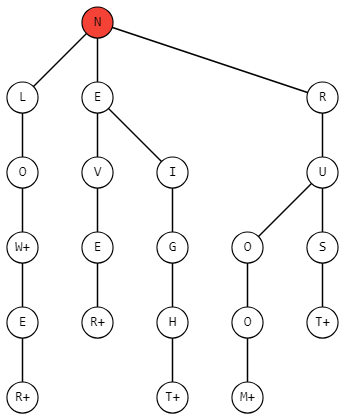


Рисунок 2 (а). Начинается поиск места под новый ключ.

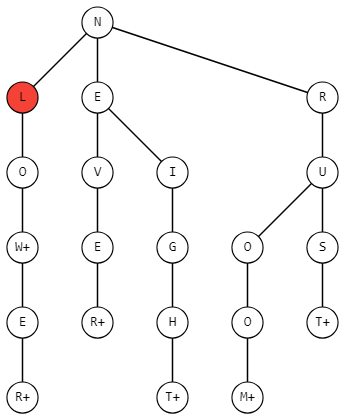


Рисунок 2 (б). Найден символ, который совпадает с первым символом ключа

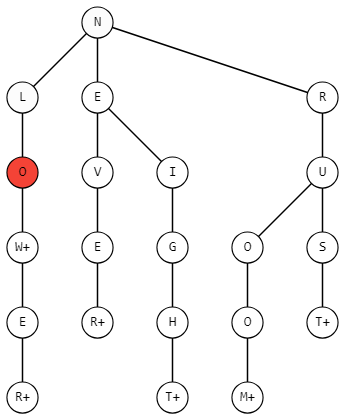


Рисунок 2 (в). Найден второй символ, совпадающий с символом ключа.

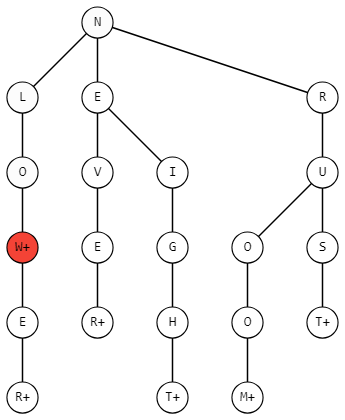


Рисунок 2 (г). Найден символ, который не совпадает с символом ключа.

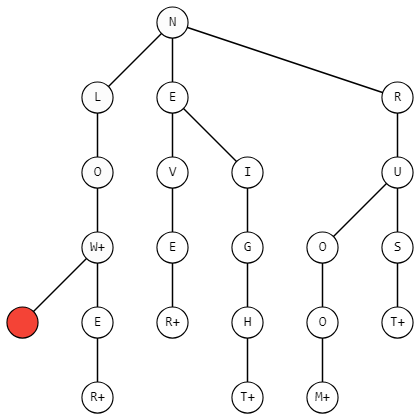


Рисунок 2 (д). Происходит смещение указателя к нижнему (отсутствующему) дочернему узлу от несовпадающего символа.

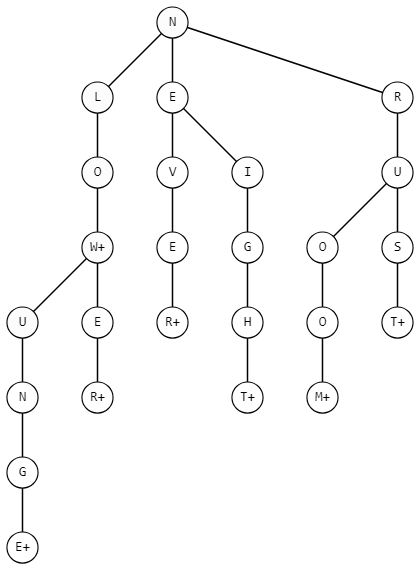


Рисунок 2 (е). Место для добавления ключа найдено, поэтому происходит выделение памяти и заполнение полей узлов. Ключ добавлен.

## 2. Удаление ключа из дерева

Удаление ключа является более сложной операцией, чем добавление. Во время удаления могут возникать различные неправильные состояния дерева, каждое из которых необходимо правильно обработать.

|  |  |
| --- | --- |
| Алгоритм 2 tst\_delete | |
| 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 | function tst\_delete(tree, key):  if tree is NULL:  return NULL  prev = NULL  indicator = NONE  node = tree  node = start\_of\_key(node, key[0], prev, indicator)  if node is NULL:  return tree  key\_trace[max\_deep]  indicators[max\_deep]  init\_key\_trace(key\_trace, indicators)  indicators[0] = indicator  node = end\_of\_key(node, key\_trace, indicators, key)  if node is NULL:  return tree  last\_sym = length of key - 1  last\_node = 0  while key\_trace[last\_node + 1] is not NULL:  last\_node = last\_node + 1  node = free\_key(node, key\_trace, indicators, last\_node, last\_sym, key)  if node.eqkid is not NULL:  return tree  if prev is NULL and not last\_node and not node.end:  tst\_restore\_prop\_any\_case(node, tree)  return tree  if node.end and not node.hikid and not node.lokid:  return tree  if last\_node:  tst\_restore\_prop(indicators[last\_node], node, key\_trace[last\_node - 1])  else:  tst\_restore\_prop(indicators[0], node, prev)  return tree |

Удаление начинается с поиска узла, в котором хранится первый символ удаляемого ключа. Это действие происходит с помощью вспомогательной операции “start\_of\_key” (все вспомогательные операции будут описаны после операции удаления). Если такой узел был найден, то он сохраняется в переменную node, при этом в переменную prev сохраняется его родительский узел (если только первый символ ключа не хранится в корне, иначе в переменной prev ничего не хранится), а в переменную indicator сохраняется его отношение к родительскому узлу. Если такой узел не был найден, значит удаляемого ключа нет в дереве и алгоритм завершает свою работу.

Далее происходит поиск всех остальных символов ключа с использованием вспомогательной операции “end\_of\_key”. Помимо поиска данная операция сохраняет в массив key\_trace все узлы, которые лежат на пути от узла, хранящего в себе первый символ ключа до узла, хранящего в себе последний символ. Некоторые узлы на пути могут хранить в себе символы, которые не являются частью ключа из-за особенностей вставки ключей. Помимо этого, в массив indicators сохраняются отношения узлов к их родительским узлам. После выполнения “end\_of\_key” в переменную node сохраняется узел, хранящий в себе последний символ ключа (если он был найден). Если такой символ не был найден, значит соответствующий ключ отсутствует в дереве и алгоритм завершает свою работу.

После того, как все узлы, относящиеся к удаляемому ключу, были найдены, начинается последовательное удаление узлов, которое происходит, c помощью операции “free\_key”, начиная с последнего найденного узла, до тех пор, пока не встретится узел, принадлежащий более чем одному ключу или пока не будет удалён весь ключ, кроме первого символа ключа (узел с первым символом требуется удалять особенным образом). Также после выполнения операции “free\_key” в last\_node и last\_sym сохраняются индекс узла родителя последнего удалённого узла и индекс символа ключа, который ещё хранится в дереве.

Проверка осуществляется следующим образом: если какой-либо узел на пути от узла, хранящего в себе первый символ ключа, к узлу, хранящему в себе последний символ, помечен как узел, хранящий в себе последний символ какого-либо ключа или имеет дочерние узлы, кроме удаляемых, или является узлом, хранящим в себе первый символ удаляемого ключа или символ, которого нет в удаляемом ключе, то удаление прекращается.

После того, как были удалены все узлы, которые не нарушали условия, в переменную node сохраняется родитель последнего удалённого узла и начинается проверка нарушений свойств дерева, которые могли возникнуть в ходе удаления.

Первое условие проверяет, нет ли у node равного дочернего узла. Если такой узел есть, значит последний удалённый узел был нижним, либо верхним дочерним узлом node. В таком случае более никаких действий не требуется и алгоритм завершает свою работу.

Следующее условие проверяет, происходит ли удаление корня при удалении ключа из дерева. Если складывается такая ситуация, значит переменная prev ничего не хранит, при этом были удалены все символы ключа, кроме первого и первый символ ключа не является завершающим символом целого ключа.

Если условие выполняется и происходит удаление корня, то вызывается вспомогательная операция “tst\_restore\_prop\_any\_case”, которая перестраивает дерево для того, чтобы обозначить новый корень и сохранить бинарные свойства дерева. Подробности этой вспомогательной операции будут описаны позже. Далее алгоритм завершает свою работу.

Если не происходит удаление корня, то проверяется тот случай, при котором узел node помечен как конец ключа и не имеет дочерних узлов. В таком случае более никаких действий не требуется и возвращается корень дерева.

После этого происходит последняя проверка с одним условием, при котором node может быть узлом, хранящим первый символ ключа, либо любой другой символ. Если node хранит в себе первый символ ключа, то происходит восстановление свойств дерева вспомогательной операцией “tst\_restore\_prop”, которая принимает на вход индикатор, узел node и его родителя из массива key\_trace. В противном случае на вход подаются значения первого элемента indicators, переменой node и prev. После чего происходит завершение операции удаления.

Процесс удаления наглядно виден на рисунке 3 на примере удаления ключа LOUNGE.

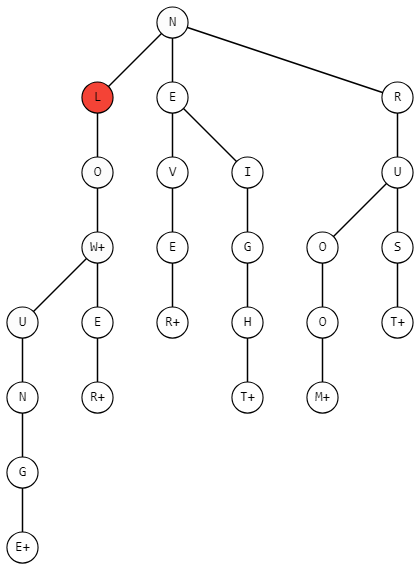


Рисунок 3 (а). Найден первый символ удаляемого ключа.

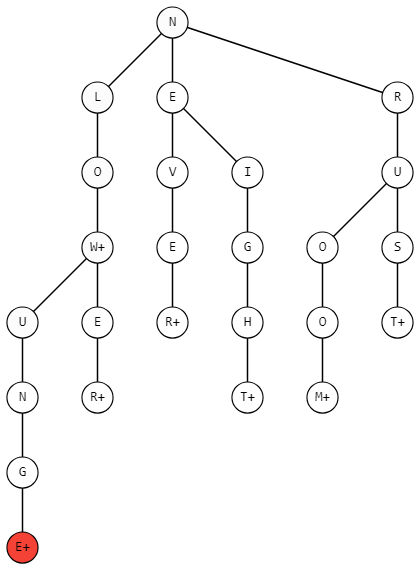


Рисунок 3 (б). Найден последний символ удаляемого ключа. Массив key\_trace при этом заполнен узлами с символами ‘L’, ‘O’, ‘W’, ‘U’, ‘N’, ‘G’, ‘E’.

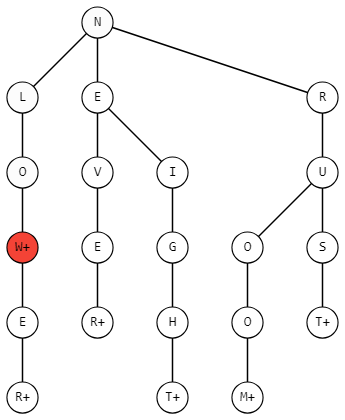


Рисунок 3 (в). Происходит последовательное удаление узлов ключа до узла с символом ‘W’, так как данный узел помечен как завершающий и имеет дочерний узел помимо удаляемого. На этом процесс удаления завершён и возвращается корень дерева.

## 2.1 Вспомогательные операции для удаления

### 2.1.1 start\_of\_key

|  |  |
| --- | --- |
| Алгоритм 3 start\_of\_key | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | function start\_of\_key(node, first\_sym, prev, indicator):  while node.ch is not equal to first\_sym:  if first\_sym > node.ch:  if node.hikid is not NULL:  indicator = HI  prev = node  node = node.hikid  else:  return NULL  else if first\_sym < node.ch:  if node.lokid is not NULL:  indicator = LO  prev = node  node = node.lokid  else:  return NULL  return node |

Данная операция принимает на вход следующие данные: node – узел с которого начнётся поиск нужного узла, first\_sym – переменная, в которой хранится значение первого символа ключа, prev – переменная, в которую будет сохранён узел, предшествующий узлу с первым символом ключа и indicator – переменная, в которой будет хранится информация об отношении найденного узла node к узлу prev.

Внутри самой операции происходит поиск наподобие того, что происходит в бинарном дереве поиска, то есть, если символ меньше того, что хранится внутри узла, то узел переходит к нижнему дочернему узлу, а если символ больше, то к верхнему дочернему. Если же нужный дочерний узел отсутствует, значит в дереве отсутствует искомый первый символ ключа и поиск завершается с возвратом NULL. А если первый символ равен символу в каком либо из узлов во время поиска, то цикл поиска завершается и возвращается указатель на найденный узел.

### 2.1.2 end\_of\_key

Данная операция нужна для того, чтобы найти все узлы, хранящие в себе символы ключа. Функция принимает на вход переменную node, хранящую в себе узел с первым символом ключа, массив key\_trace, в котором будут хранится указатели на все узлы на пути от узла с первым символом до узла с последним символом, включая крайние узлы, массив indicators, который хранит информацию об отношении узлов из массива key\_trace к их родительским узлам и сам ключ key.

|  |  |
| --- | --- |
| Алгоритм 4 end\_of\_key | |
| 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 | function end\_of\_key(node, key\_trace, indicators, key):  indicator = indicators[0]  key\_is\_part = true  n\_symbol = 0  for i from 0 to max\_deep:  key\_trace[i] = node  indicators[i] = indicator  key\_is\_part = (node.ch == key[n\_symbol])  if key\_is\_part:  n\_symbol = n\_symbol + 1  while not key\_is\_part:  if key[n\_symbol] > node.ch:  if node.hikid is not NULL:  node = node.hikid  indicator = HI  i = i + 1  key\_trace[i] = node  indicators[i] = indicator  else:  return NULL  else if key[n\_symbol] < node.ch:  if node.lokid is not NULL:  node = node.lokid  indicator = LO  i = i + 1  key\_trace[i] = node  indicators[i] = indicator  else:  return NULL  else:  n\_symbol = n\_symbol + 1  key\_is\_part = true  if key[n\_symbol] is NULL:  if node.end:  return node  else:  return NULL  if node.eqkid is not NULL:  node = node.eqkid  indicator = EQ  else:  return NULL  return NULL |

В самом начале функции переменной indicator присваивается значение первого элемента массива indicators, в котором к началу выполнения функции уже должна хранится информация об отношении узла node к его родительскому узлу. Также объявляется переменная key\_is\_part, которая понадобится для проверки принадлежности символа, хранящегося в каком-либо узле, к удаляемому ключу. Помимо этого, происходит объявление переменной n\_symbol, которая используется для обращения к символам ключа по индексу.

Далее начинается цикл для переменной i от нуля до значения внешней переменной max\_deep, которая используется при объявлении массивов key\_trace и indicators и задаёт их размер. Предполагается, что такого размера массивов будет достаточно. В начале цикла в массив key\_trace сохраняется узел node и его индикатор. Далее происходит проверка принадлежности символа ключа с индексом n\_symbol к узлу node. Если символ находится в узле, то происходит переход к следующему символу, то есть значение переменной n\_symbol увеличивается на единицу. А если символ в узле не соответствует символу ключа, то происходит поиск наподобие того, что в функции start\_of\_key, но при этом все узлы, которые появляются на протяжении поиска, сохраняются в массив key\_trace. Если символ не найден в дереве, то функция возвращает NULL, а если найден, то n\_symbol увеличивается на единицу и поиск завершается.

После поиска происходит две проверки. Сначала проверяется, есть ли ещё символы в ключе, и если их нет и при этом узел node помечен как завершающий, то функция заканчивает свою работу и возвращает узел node. Если же узел node не помечен как завершающий, значит ключ не найден, и функция возвращает NULL. А если в ключе ещё остались символы, то проверяется наличие равного дочернего узла node и переход к нему, если он есть, либо завершение с NULL, если его нет. После цикл для i переходит к следующей итерации. Если цикл дошёл до конца и завершил свою работу, значит в массиве key\_trace закончилось место, чего не должно произойти. В таком случае функция вернёт NULL.

### 2.1.3 free\_key

Следующая операция удаляет все узлы, которые получится, из тех, что были найдены в ходе работы функции end\_of\_key. Функция принимает на вход node (указатель не узел, хранящий в себе последний символ ключа, массив узлов key\_trace, массив индикаторов indicators, переменные last\_node, которая хранит в себе индекс последнего узла в массиве key\_trace, last\_sym, которая хранит индекс последнего символа в ключе и сам ключ key.

В самом начале работы функции узел node помечается как узел, который не хранит завершающий символ, после чего начинается цикл с множеством условий. Цикл выполняется пока у node нет дочерних узлов, узел node не помечен как узел с последним символом ключа, узел node не является первым символом ключа и узел node хранит в себе тот символ, который является частью удаляемого ключа.

|  |  |
| --- | --- |
| Алгоритм 5 free\_key | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function free\_key(node, key\_trace, indicators, last\_node, last\_sym, key):  node.end = false  while (not node.eqkid and not node.hikid and not node.lokid and not node.end  and last\_node and node.ch == key[last\_sym]):  free(node)  last\_node = last\_node - 1  last\_sym = last\_sym - 1  node = key\_trace[last\_node]  case indicators[last\_node + 1]:  EQ: node.eqkid = NULL  HI: node.hikid = NULL  LO: node.lokid = NULL  return node |

В цикле происходит освобождение памяти, выделенной под узел node, после чего значение переменных last\_node и last\_sym уменьшается на единицу и узлу node присваивается адрес его родителя. Далее соответствующему полю узла node при помощи массива indicators присваивается значение NULL, чтобы удалить узел из дерева окончательно. После завершения цикла происходит возвращение узла node.

### 2.1.4 tst\_restore\_prop\_any\_case

Эта функция восстанавливает свойства дерева в тех случая, когда удаляемый узел их нарушает. Функция принимает на вход переменную node, которая хранит адрес узла, который нужно удалить, и переменную replacement, которая хранит в себе адрес того места в дереве, в котором требуется перестроить дерево после удаления узла node.

Функция начинается с проверки того, какие дочерние узлы имеются у узла node. Если у него есть верхний дочерний узел, то он встаёт на место node в переменную replacement, после этого происходит проверка наличия нижнего дочернего узла node. Если такого узла нет, то дерево перестроено и функция завершает свою работу, если такой узел есть, то его необходимо сделать нижним дочерним узлом меньшего узла в верхнем поддерево node. Для этого нижний дочерний узел node сохраняется в переменную lo\_node, после чего происходит поиск наименьшего узла в верхнем поддереве node. Поиск происходит следующим образом: в переменную hi\_node сохраняется указатель на верхний дочерний узел node, узел hi\_node заменяется на нижний дочерний узел до тех пор, пока такой имеется. После того, как был найден отсутствующий нижний дочерний узел, им становится тот узел, который был сохранён в переменную lo\_node. После чего дерево перестроено, и функция успешно завершает свою работу.

|  |  |
| --- | --- |
| Алгоритм 6 tst\_restore\_prop\_any\_case | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | function tst\_restore\_prop\_any\_case(node, replacement):  if node.hikid is not NULL:  replacement = node.hikid  lo\_node = NULL  if node.lokid is not NULL:  lo\_node = node.lokid  else:  free(node)  return  hi\_node = node.hikid  while hi\_node.lokid is not NULL:  hi\_node = hi\_node.lokid  hi\_node.lokid = lo\_node  free(node)  else if node.lokid is not NULL:  replacement = node.lokid  free(node)  else:  free(node)  replacement = NULL |

Если же у node нет верхнего дочернего узла, то происходит проверка на наличие нижнего дочернего узла. Если такой есть, то он просто встаёт на место узла node в переменной replacement и функция завершает свою работу.

Если у node не было дочерних узлов, то переменной replacement просто присваивается NULL.

### 2.1.5 tst\_restore\_prop

Данная функция является расширением для предыдущей. Она принимает на вход indicator, node и prev (родительский узел node). После чего она вызывает функцию tst\_restore\_prop\_any\_case, используя в качестве второго аргумента один из дочерних узлов prev, в зависимости от того, что хранится в переменной indicator.

|  |  |
| --- | --- |
| Алгоритм 7 tst\_restore\_prop | |
| 1  2  3  4  5 | function tst\_restore\_prop(indicator, node, prev):  switch indicator:  case LO: tst\_restore\_prop\_any\_case(node, prev.lokid)  case HI: tst\_restore\_prop\_any\_case(node, prev.hikid)  case EQ: tst\_restore\_prop\_any\_case(node, prev.eqkid) |

## 3. Удаление всего дерева

Для удаления дерева применяется рекурсивная функция, которая принимает на вход корень дерева и вызывает саму себя для его дочерних узлов, если такие имеются. В конце функции происходит освобождение памяти, выделенной для каждого узла.

|  |  |
| --- | --- |
| Алгоритм 8 tst\_delete\_tree | |
| 1  2  3  4  5  6  7  8  9  10 | function tst\_delete\_tree(tree):  if tree is NULL:  return  if tree.eqkid is not NULL:  tst\_delete\_tree(tree.eqkid)  if tree.lokid is not NULL:  tst\_delete\_tree(tree.lokid)  if tree.hikid is not NULL:  tst\_delete\_tree(tree.hikid)  free(tree) |

## 4. Вывод списка слов

### 4.1 tst\_print\_all\_words

Для вывода всех слов, которые хранятся в дереве применяется функция tst\_print\_all\_words, которая принимает на вход корень дерева.

Внутри функции создаётся строка размером max\_deep для буферизации слова, которое нужно будет вывести. После чего в строку word записывается первый символ, который лежит в корне дерева, и если корень дерева является узлом, помеченным, как завершающий, то word тут же выводится.

Далее, если у корня есть равный дочерний узел, то вызывается функция tst\_print\_words\_from\_node, которая выводит все завершения префикса. В качестве префикса этой функции на вход поступает единственный символ, который хранится в текущем узле.

|  |  |
| --- | --- |
| Алгоритм 9 tst\_print\_all\_words | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | function tst\_print\_all\_words(tree):  if tree is not NULL:  word[max\_deep]  word[0] = NULL  symbol = tree.ch  append symbol to word  if tree.end:  print word  if tree.eqkid is not NULL:  tst\_print\_words\_from\_node(tree.eqkid, word, EQ)  else:  print "Here is no tree"  if tree.lokid is not NULL:  tst\_print\_all\_words(tree.lokid)  if tree.hikid is not NULL:  tst\_print\_all\_words(tree.hikid) |

После этого, если у корня есть нижний или верхний дочерний узлы, то функция tst\_print\_all\_words рекурсивно вызывается для этих узлов и процесс повторяется до тех пор, пока не закончатся все узлы, хранящие в себе первые символы слов.

### 4.2 tst\_print\_words\_from\_node

На вход функция принимает префикс word, узел node, который является дочерним узлом по отношению к тому, который хранит в себе последний символ префикса и переменную indicator, которая хранит в себе отношение узла node к его родительскому узлу.

|  |  |
| --- | --- |
| Алгоритм 10 tst\_print\_words\_from\_node | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function tst\_print\_words\_from\_node(node, word, indicator):  word\_copy = copy of word  if indicator is LO or indicator is HI:  word\_copy[length of word\_copy - 1] = NULL  symbol = node.ch  append symbol to word\_copy  if node.end:  print word\_copy  if node.eqkid is not NULL:  tst\_print\_words\_from\_node(node.eqkid, word\_copy, EQ)  if node.lokid is not NULL:  tst\_print\_words\_from\_node(node.lokid, word\_copy, LO)  if node.hikid is not NULL:  tst\_print\_words\_from\_node(node.hikid, word\_copy, HI) |

Данная функция является рекурсивной, поэтому в начале функции создаётся копия префикса в строку word\_copy. После этого происходит проверка индикатора. Если node является верхним или нижним дочерним узлом своего родителя, значит последний символ word не является частью ключа. Это обусловлено особенностью добавления ключей, которая описывалась выше.

После этого к word\_copy в конец дописывается символ, хранящийся в узле node и происходит проверка на наличие метки завершения ключа у узла node. Если она есть, то word\_copy выводится.

Далее функция вызывает саму себя для всех дочерних узлов node, если они есть. При этом в качестве префикса в качестве аргумента используется word\_copy. Также используется индикатор, соответствующий дочернему узлу.

## 5. Поиск

Так как дерево не хранит в себе ничего, кроме ключей, а ключи при этом хранятся в разных узлах, было принято решение утроить поиск так, чтобы по итогу возвращалось значение true в случае, если ключ найден и значение false, если искомого ключа нет в дереве.

### 5.1 tst\_lookup

Данная функция занимается поиском ключа и принимает на вход указатель на корень дерева tree и ключ key.

|  |  |
| --- | --- |
| Алгоритм 11 tst\_lookup | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | function tst\_lookup(tree, key):  if tree is NULL:  return false  tree = tst\_find\_sym(tree, key[0])  if tree is NULL:  return false  key = key + 1  while key[0] is not NULL:  if tree.eqkid is not NULL:  if tree.eqkid.ch is equal to key[0]:  tree = tree.eqkid  key = key + 1  continue  tree = tree.eqkid  else:  return false  tree = tst\_find\_sym(tree, key[0])  if tree is NULL:  return false  key = key + 1  return tree.end |

Функция использует вспомогательную функцию tst\_find\_sym для того, чтобы правильно искать символы в дереве. Сначала при помощи этой функции происходит поиск первого символа ключа в дереве. После этого происходит переход к следующему символу и начинается цикл до конца ключа, если первый символ был найден в дереве.

Цикл начинается с проверки наличия равного дочернего узла у tree, где к моменту начала цикла сохранён узел с первым символом. Если такой дочерний узел есть, то происходит проверка, совпадает ли его символ с символом ключа. Если это так, то в tree записывается равный дочерний узел и происходит переход к следующему символу ключа, а цикл переходит к следующей итерации.

Если символ, который хранится в равном дочернем узле не совпадает с символом ключа, то в tree записывается адрес равного дочернего узла и цикл продолжает свою работу поиском символа ключа в дереве. Если символ найден, то произойдёт переход к следующему символу и новой итерации. Если символ не найден, то вернётся false. То же самое значение вернётся, если у tree отсутствует равный дочерний узел в начале цикла.

Если цикл завершил свою работу, значит для каждого символа ключа был найден соответствующий узел. Тогда функция возвращает метку последнего узла.

### 5.2 tst\_find\_sym

|  |  |
| --- | --- |
| Алгоритм 12 tst\_find\_sym | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | function tst\_find\_sym(node, symbol):  if node is NULL:  return NULL  while symbol is not equal to node.ch:  if symbol < node.ch:  if node.lokid is not NULL:  node = node.lokid  else:  return NULL  else if symbol > node.ch:  if node.hikid is not NULL:  node = node.hikid  else:  return NULL  return node |

Функция принимает на вход указатель на узел node и символ ключа symbol. Она применяется в тех случаях, когда символ не найден в узле node и необходимо проверить его наличие в нижнем и верхнем поддереве узла node. Внутри функции происходит обычный поиск, подобие которого уже описывалось выше.

## 6. Префиксный поиск

Помимо обычного поиска, в рамках данной курсовой работы был реализован так же и префиксный поиск. Префиксный поиск позволяет найти все завершения n первых символов ключа (n может равняться количеству символов в ключе).

### tst\_prefix\_search

Данная функция принимает на вход указатель на корень дерева и префикс.

|  |  |
| --- | --- |
| Алгоритм 13 tst\_prefix\_search | |
| 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 | function tst\_prefix\_search(tree, prefix):  if tree is NULL:  print "Here is no keys"  return  prefix\_copy = copy of prefix  tree = tst\_find\_sym(tree, prefix[0])  if tree is NULL:  print "Here is no keys with prefix '", prefix\_copy, "'"  return  prefix = prefix + 1  while prefix[0] is not NULL:  if tree.eqkid is not NULL:  if tree.eqkid.ch is equal to prefix[0]:  tree = tree.eqkid  prefix = prefix + 1  continue  tree = tree.eqkid  else:  print "Here is no keys with prefix '", prefix\_copy, "'"  return  tree = tst\_find\_sym(tree, prefix[0])  if tree is NULL:  print "Here is no keys with prefix '", prefix\_copy, "'"  return  prefix = prefix + 1  if tree.end:  print prefix\_copy  if tree.eqkid is not NULL:  tst\_print\_words\_from\_node(tree.eqkid, prefix\_copy, EQ) |

Первым делом создаётся копия префикса, которая понадобится в дальнейшем. После этого происходит поиск первого символа префикса в дереве. Далее происходит переход к следующему символу префикса и поиск остальных его символов в цикле. Данный поиск осуществляется так же, как и в функции tst\_lookup. Если найдены все символы префикса, то происходит вывод префикса, если узел с последним символом префикса помечен, как завершающий, после чего вызывается вспомогательная функция tst\_print\_words\_from\_node, которая выводит все завершения этого префикса, которые есть в дереве.

# АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ

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

Одной из особенностью тернарного дерева поиска является то, что оно способно накапливать потенциал при добавлении большого количества ключей. Изначально, при добавлении первых ключей, скорость увеличения количества новых узлов значительно выше, чем скорость увеличения количества ключей в дереве. Лучше всего это рассмотреть на конкретном примере. Далее будут описываться эксперименты, при которых в дерево добавляется 1 000 000 ключей, состоящих из 10 символов. Если добавлять ключи в случайном порядке, то при добавлении первых ключей в дерево для каждого нового ключа в среднем необходимо добавлять 10 новых узлов. То есть скорость роста количества ключей в 10 раз выше скорости роста количества узлов. Но при дальнейшем добавлении в дереве становится всё больше префиксов, что облегчает процесс добавления ключа, так как для каждого нового ключа требуется добавлять всё меньше новых узлов. При этом, если после добавления 1 000 000 ключей из 10 символов начать добавлять ключи из меньшего числа символов, то для многих ключей и вовсе не потребуется добавлять новые узлы, так как они целиком будут являться префиксами уже добавленных узлов, и всё, что надо будет сделать, это просто пометить существующий узел, как завершающий. Вместе со снижением скорости увеличения количества новых узлов снижается так же и скорость роста высоты дерева. Далее будет объяснено, почему это имеет значение.

## Поиск ключа

Длительность выполнения операции поиска напрямую зависит от того, сколько узлов понадобится пройти от корня дерева до самого высокого узла с последним символом. Таким образом, в худшем случае, сложность операции поиска составляет O(H), где H – высота дерева.

Но если построить зависимость между временем выполнения операции и количеством ключей в дереве, то можно провести амортизационный анализ алгоритма, при котором усредняется время поиска всех ключей, добавленных в дерево.

При таком подходе, амортизированное время поиска в худшем случае будет составлять приблизительно O(N + K), где N – количество ключей в дереве, а K – длина искомого ключа. Худший случай возникает в тот момент, когда ключи в дерево добавляются упорядоченно. При таком добавлении скорость увеличения количества новых узлов будет замедляться по мере добавления новых ключей с одинаковым первым символом и резко возрастать каждый раз, когда будет появляться новый первый символ. Помимо этого, в таком случае каждое поддерево вырождается в связный список, что также сказывается на эффективности, так как негативно влияет на высоту дерева. Экспериментальное исследование худшего случая можно увидеть на рисунке 4 (а).

Лучший случай возникает в тот момент, когда ключи в дерево добавляются в случайном порядке. В этом случае амортизированное время поиска будет O(log(N) + K). Экспериментальное исследование лучшего случая изображено на рисунке 4 (б).

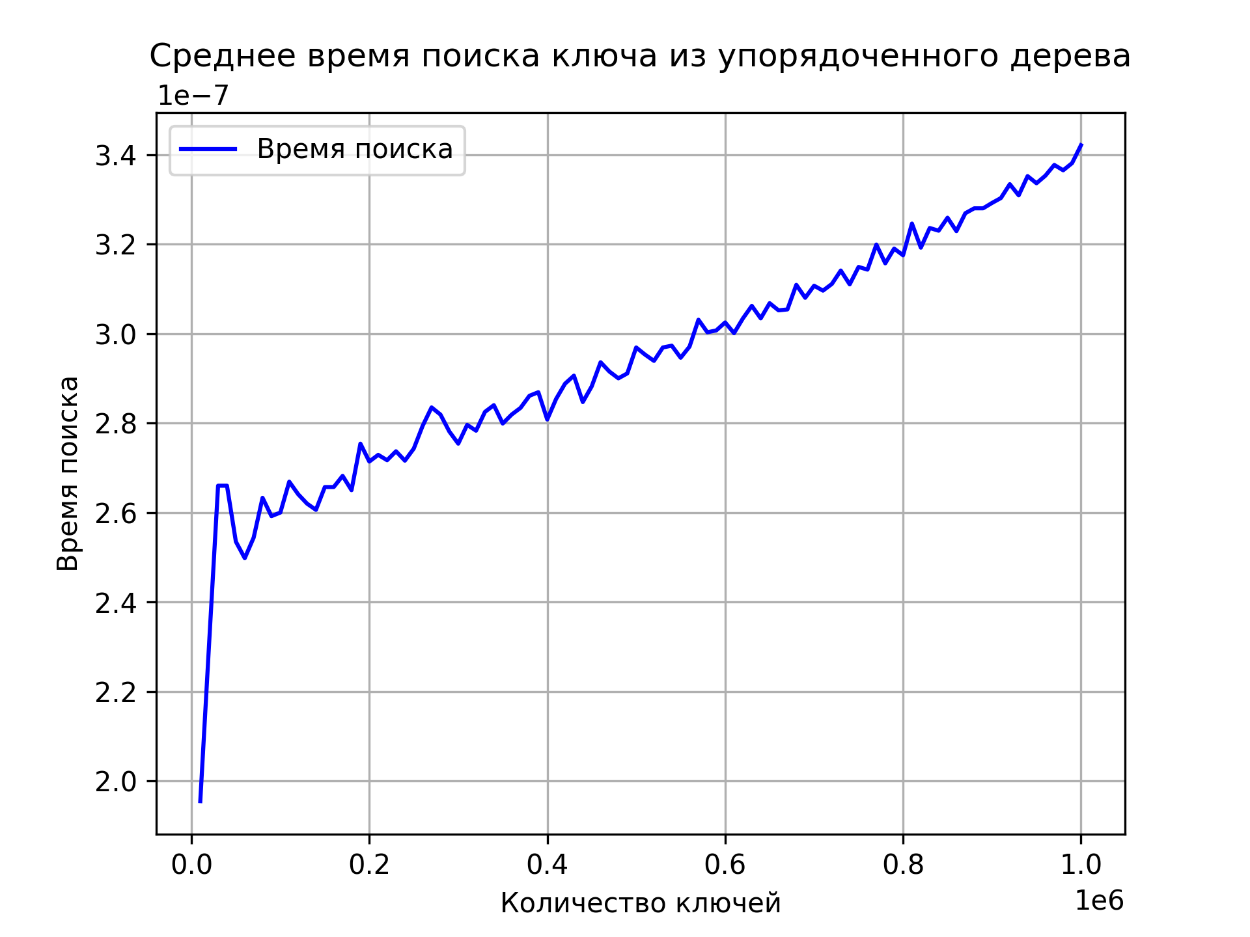


Рисунок 4 (а). Время поиска ключа из упорядоченного дерева.

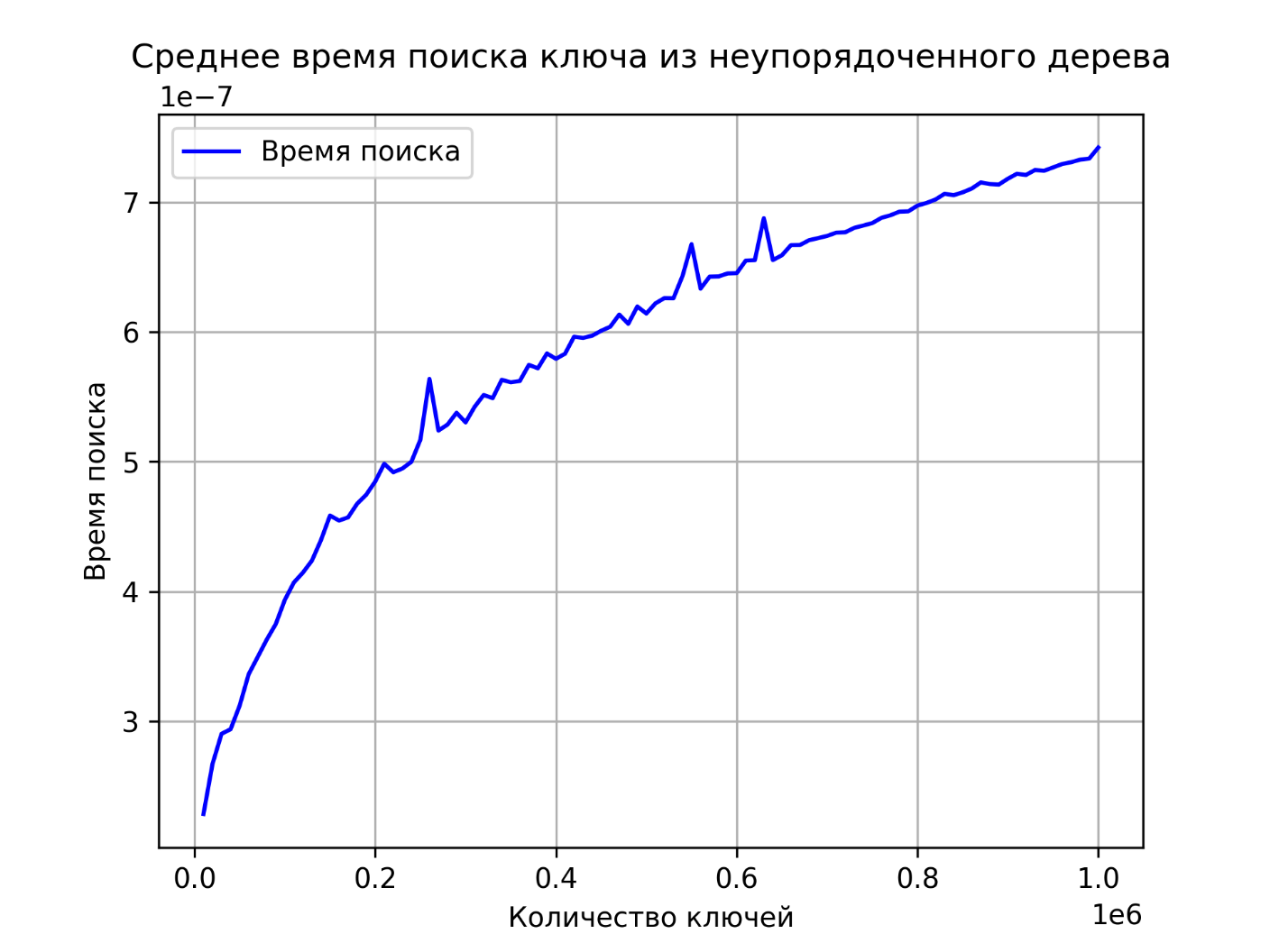


Рисунок 4 (б). Время поиска ключа из неупорядоченного дерева.

## Добавление ключей в дерево

При выполнении каждой операции добавления каждый раз происходит поиск того места, с которого начинается добавление недостающих узлов. Этот поиск практически ничем не отличается от самой операции поиска, поэтому амортизированное время в худшем случае составит O(N + P + S), где P – длина уже имеющегося префикса, а S – длина недостающего суффикса. Длины как суффикса, так и префикса могут равняться длине добавляемого ключа. В среднем случае, при неупорядоченном добавлении, амортизированное время будет O(log(N) + P + S). Экспериментальное подтверждение анализа наглядно изображено на рисунке 5.



Рисунок 5 (а). Удаление ключей из неупорядоченного дерева.



Рисунок 5 (б). Удаление ключей из упорядоченного дерева.

## Удаление ключа из дерева

Операция удаления является обратной к операции добавления. При выполнении этой операции также производится поиск, но после поиска узлы не добавляются, а удаляются. Так же, иногда при удалении приходится перестраивать дерево для восстановления его свойств. При этом в худшем случае требуется сделать D – 2 шага по узлам дерева, где D – это мощность алфавита. Таким образом, амортизированное время худшего случая составляет O(N + S + D), где S – длина удаляемого суффикса, которая может равняться длине ключа.

В среднем случае дерево формируется равномерно, и тогда амортизированное время удаления будет O(log(N) + S + log(D)). Графики сложности алгоритма проиллюстрированы на рисунке 6.



Рисунок 6 (а). Удаление из неупорядоченного дерева.



Рисунок 6 (б). Удаление из упорядоченного дерева.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы была реализована и детально проанализирована такая структура данных как тернарное дерево поиска. Несмотря на то, что эта структура данных может иметь множество различных применений, большого распространения она не получила.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. URL: <https://en.wikipedia.org/wiki/Ternary_search_tree>

# ПРИЛОЖЕНИЕ

Исходный код программы

main.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 | #include "experimental\_funcs.h"  #include "tst.h"  int main()  {  srand(time(0));  char input = 0;  tst\* tree = NULL;  char string[128];  double time;  while (input != '0') {  printf("It's my super program to demonstrate ternary search tree\n");  printf("Choose option or press 0 to exit\n");  printf("(Press 'Enter' for the second time after every operation)\n");  printf("Option list:\n");  printf("1. Create new tree (your old tree will be deleted)\n");  printf("2. Insert key to your tree\n");  printf("3. Print all inserted keys\n");  printf("4. Delete key\n");  printf("5. Load keys to tree from file\n");  printf("6. Make prefix search\n");  printf("7. Update experimental data\n");  input = getchar();  while (getchar() != '\n')  ;  switch (input) {  case '0':  tst\_delete\_tree(tree);  break;  case '1':  time = wtime();  tst\_delete\_tree(tree);  tree = NULL;  printf("All tree was deleted in %.7lf seconds\n", wtime() - time);  getchar();  break;  case '2':  printf("Enter key: ");  fgets(string, 100, stdin);  string[strlen(string) - 1] = '\0';  time = wtime();  tree = tst\_insert(tree, string);  printf("Key '%s' was added in %.7lf seconds\n",  string,  wtime() - time);  getchar();  break;  case '3':  time = wtime();  tst\_print\_all\_words(tree);  printf("All keys were printed in %.7lf seconds\n", wtime() - time);  getchar();  break;  case '4':  printf("Enter key: ");  fgets(string, 100, stdin);  string[strlen(string) - 1] = '\0';  time = wtime();  tree = tst\_delete(tree, string);  printf("Key '%s' was deleted in %.7lf seconds\n",  string,  wtime() - time);  getchar();  break;  case '5':  printf("Enter filename: ");  fgets(string, 100, stdin);  string[strlen(string) - 1] = '\0';  FILE\* file = fopen(string, "r");  if (file == NULL) {  printf("Can't open file :(\n");  getchar();  break;  }  char key[128];  int i = 0;  time = wtime();  while (fscanf(file, "%s", key) != EOF) {  tree = tst\_insert(tree, key);  i++;  }  printf("%d keys were added to the tree in %.7lf seconds\n",  i,  wtime() - time);  fclose(file);  getchar();  break;  case '6':  printf("Enter the prefix: ");  fgets(string, 100, stdin);  string[strlen(string) - 1] = '\0';  time = wtime();  tst\_prefix\_search(tree, string);  printf("All keys were finded in %.7lf seconds\n", wtime() - time);  getchar();  break;  case '7':  time = wtime();  deleting\_from\_last\_time\_random\_experiment();  deleting\_from\_last\_time\_sorted\_experiment();  random\_insertion\_time\_experiment();  sorted\_insertion\_time\_experiment();  random\_lookup\_time\_random\_experiment();  random\_lookup\_time\_sorted\_experiment();  last\_lookup\_time\_random\_experiment();  last\_lookup\_time\_sorted\_experiment();  printf("Data has been updated in %f seconds\n\n", wtime() - time);  getchar();  break;  }  }  } |

tst.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  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419 | #include "tst.h"  int max\_deep = 128;  tst\* create\_node(char ch)  {  tst\* node = malloc(sizeof(tst));  if (!node)  return NULL;  node->lokid = NULL;  node->eqkid = NULL;  node->hikid = NULL;  node->ch = ch;  node->end = false;  return node;  }  tst\* tst\_insert(tst\* tree, char\* key)  {  tst \*node = tree, \*prev = NULL;  int indicator = NONE;  while (node != NULL) {  if (\*key < node->ch) {  indicator = LO;  prev = node;  node = node->lokid;  } else if (\*key > node->ch) {  indicator = HI;  prev = node;  node = node->hikid;  } else {  indicator = EQ;  prev = node;  node = node->eqkid;  key++;  if (\*key == '\0') {  prev->end = true;  return tree;  }  }  }  node = create\_node(\*key);  if (tree == NULL)  tree = node;  switch (indicator) {  case NONE:  break;  case LO:  prev->lokid = node;  break;  case HI:  prev->hikid = node;  break;  case EQ:  prev->eqkid = node;  break;  }  prev = node;  key++;  while (\*key != '\0') {  node = create\_node(\*key);  key++;  prev->eqkid = node;  prev = node;  }  node->end = true;  return tree;  }  tst\* start\_of\_key(tst\* node, char first\_sym, tst\*\* prev, int\* indicator)  {  while (node->ch != first\_sym) {  if (first\_sym > node->ch) {  if (node->hikid) {  \*indicator = HI;  \*prev = node;  node = node->hikid;  } else {  return NULL;  }  }  if (first\_sym < node->ch) {  if (node->lokid) {  \*indicator = LO;  \*prev = node;  node = node->lokid;  } else {  return NULL;  }  }  }  return node;  }  tst\* end\_of\_key(tst\* node, tst\*\* key\_trace, int\* indicators, char\* key)  {  int indicator = indicators[0];  bool key\_is\_part = 1;  int n\_symbol = 0;  for (int i = 0; i < max\_deep; i++) {  key\_trace[i] = node;  indicators[i] = indicator;  key\_is\_part = node->ch == key[n\_symbol] ? 1 : 0;  if (key\_is\_part)  n\_symbol++;  while (!key\_is\_part) {  if (key[n\_symbol] > node->ch) {  if (node->hikid) {  node = node->hikid;  indicator = HI;  i++;  key\_trace[i] = node;  indicators[i] = indicator;  } else {  return NULL;  }  } else if (key[n\_symbol] < node->ch) {  if (node->lokid) {  node = node->lokid;  indicator = LO;  i++;  key\_trace[i] = node;  indicators[i] = indicator;  } else {  return NULL;  }  } else {  n\_symbol++;  key\_is\_part = true;  }  }  if (key[n\_symbol] == '\0') {  if (node->end)  return node;  else  return NULL;  }  if (node->eqkid) {  node = node->eqkid;  indicator = EQ;  } else {  return NULL;  }  }  return NULL;  }  tst\* free\_key(  tst\* node,  tst\*\* key\_trace,  int\* indicators,  int\* last\_node,  int\* last\_sym,  char\* key)  {  node->end = false;  while (!node->eqkid && !node->hikid && !node->lokid && !node->end  && \*last\_node && node->ch == key[\*last\_sym]) {  free(node);  \*last\_node -= 1;  \*last\_sym -= 1;  node = key\_trace[\*last\_node];  switch (indicators[\*last\_node + 1]) {  case EQ:  node->eqkid = NULL;  break;  case HI:  node->hikid = NULL;  break;  case LO:  node->lokid = NULL;  break;  }  }  return node;  }  void tst\_restore\_prop\_any\_case(tst\* node, tst\*\* replacement)  {  if (node->hikid) {  \*replacement = node->hikid;  tst\* lo\_node = NULL;  if (node->lokid)  lo\_node = node->lokid;  else {  free(node);  return;  }  tst\* hi\_node = node->hikid;  while (hi\_node->lokid)  hi\_node = hi\_node->lokid;  hi\_node->lokid = lo\_node;  free(node);  return;  }  if (node->lokid) {  \*replacement = node->lokid;  free(node);  return;  }  free(node);  \*replacement = NULL;  }  void tst\_restore\_prop(int indicator, tst\* node, tst\* prev)  {  switch (indicator) {  case LO:  tst\_restore\_prop\_any\_case(node, &(prev->lokid));  break;  case HI:  tst\_restore\_prop\_any\_case(node, &(prev->hikid));  break;  case EQ:  tst\_restore\_prop\_any\_case(node, &(prev->eqkid));  break;  }  }  void init\_key\_trace(tst\*\* key\_trace, int\* indicators)  {  for (int i = 0; i < max\_deep; i++) {  key\_trace[i] = NULL;  indicators[i] = NONE;  }  }  tst\* tst\_delete(tst\* tree, char\* key)  {  if (tree == NULL)  return NULL;  tst\* prev = NULL;  int indicator = NONE;  tst\* node = tree;  node = start\_of\_key(node, \*key, &prev, &indicator);  if (node == NULL)  return tree;  tst\* key\_trace[max\_deep];  int indicators[max\_deep];  init\_key\_trace(key\_trace, indicators);  indicators[0] = indicator;  node = end\_of\_key(node, key\_trace, indicators, key);  if (node == NULL)  return tree;  int last\_sym = strlen(key) - 1;  int last\_node = 0;  while (key\_trace[last\_node + 1] != NULL)  last\_node++;  node = free\_key(node, key\_trace, indicators, &last\_node, &last\_sym, key);  if (node->eqkid)  return tree;  if (prev == NULL && !last\_node && !node->end) {  tst\_restore\_prop\_any\_case(node, &tree);  return tree;  }  if (node->end && !node->hikid && !node->lokid)  return tree;  if (last\_node)  // if (node->ch == key[last\_sym] && last\_node) {  tst\_restore\_prop(indicators[last\_node], node, key\_trace[last\_node - 1]);  else  tst\_restore\_prop(indicators[0], node, prev);  return tree;  }  void tst\_print\_words\_from\_node(tst\* node, char\* word, int indicator)  {  char word\_copy[max\_deep];  strcpy(word\_copy, word);  if (indicator == LO || indicator == HI)  word\_copy[strlen(word\_copy) - 1] = '\0';  char symbol[2] = {node->ch, '\0'};  strcat(word\_copy, symbol);  if (node->end)  printf("%s\n", word\_copy);  if (node->eqkid)  tst\_print\_words\_from\_node(node->eqkid, word\_copy, EQ);  if (node->lokid)  tst\_print\_words\_from\_node(node->lokid, word\_copy, LO);  if (node->hikid)  tst\_print\_words\_from\_node(node->hikid, word\_copy, HI);  }  void tst\_print\_all\_words(tst\* tree)  {  if (tree) {  char word[max\_deep];  word[0] = '\0';  char symbol[2] = {tree->ch, '\0'};  strcat(word, symbol);  if (tree->end)  printf("%s\n", word);  if (tree->eqkid)  tst\_print\_words\_from\_node(tree->eqkid, word, EQ);  } else {  printf("Here is no tree\n");  return;  }  if (tree->lokid)  tst\_print\_all\_words(tree->lokid);  if (tree->hikid)  tst\_print\_all\_words(tree->hikid);  }  void tst\_delete\_tree(tst\* tree)  {  if (tree == NULL)  return;  if (tree->eqkid)  tst\_delete\_tree(tree->eqkid);  if (tree->lokid)  tst\_delete\_tree(tree->lokid);  if (tree->hikid)  tst\_delete\_tree(tree->hikid);  free(tree);  }  tst\* tst\_find\_sym(tst\* node, char symbol)  {  if (node == NULL)  return NULL;  while (symbol != node->ch) {  if (symbol < node->ch) {  if (node->lokid) {  node = node->lokid;  continue;  } else {  return NULL;  }  }  if (symbol > node->ch) {  if (node->hikid) {  node = node->hikid;  continue;  } else {  return NULL;  }  }  }  return node;  }  bool tst\_lookup(tst\* tree, char\* key)  {  if (tree == NULL)  return false;  tree = tst\_find\_sym(tree, \*key);  if (tree == NULL)  return false;  key++;  while (\*key != '\0') {  if (tree->eqkid) {  if (tree->eqkid->ch == \*key) {  tree = tree->eqkid;  key++;  continue;  }  tree = tree->eqkid;  } else {  return false;  }  tree = tst\_find\_sym(tree, \*key);  if (tree == NULL)  return false;  key++;  }  return tree->end;  }  void tst\_prefix\_search(tst\* tree, char\* prefix)  {  if (tree == NULL) {  printf("Here is no keys\n");  return;  }  char prefix\_copy[max\_deep];  strcpy(prefix\_copy, prefix);  tree = tst\_find\_sym(tree, \*prefix);  if (tree == NULL) {  printf("Here is no keys with prefix '%s'\n", prefix\_copy);  return;  }  prefix++;  while (\*prefix != '\0') {  if (tree->eqkid) {  if (tree->eqkid->ch == \*prefix) {  tree = tree->eqkid;  prefix++;  continue;  }  tree = tree->eqkid;  } else {  printf("Here is no keys with prefix '%s'\n", prefix\_copy);  return;  }  tree = tst\_find\_sym(tree, \*prefix);  if (tree == NULL) {  printf("Here is no keys with prefix '%s'\n", prefix\_copy);  return;  }  prefix++;  }  if(tree->end)  printf("%s\n", prefix\_copy);  if(tree->eqkid)  tst\_print\_words\_from\_node(tree->eqkid, prefix\_copy, EQ);  } |

tst.h

|  |  |
| --- | --- |
| 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 | #pragma once  #include <stdbool.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  enum { NONE = -1, LO, HI, EQ };  typedef struct tst {  struct tst\* lokid;  struct tst\* eqkid;  struct tst\* hikid;  char ch;  bool end;  } tst;  extern int max\_deep;  tst\* create\_node(char ch);  tst\* tst\_insert(tst\* tree, char\* key);  tst\* tst\_delete(tst\* tree, char\* key);  void tst\_print\_all\_words(tst\* tree);  void tst\_delete\_tree(tst\* tree);  bool tst\_lookup(tst\* tree, char\* key);  void tst\_prefix\_search(tst\* tree, char\* prefix); |

experimental\_funcs.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  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307 | #include "experimental\_funcs.h"  #include "tst.h"  int n\_words = 1000000;  int step = 10000;  int n\_iterations = 1000000;  void random\_insertion\_time\_experiment()  {  int tmp\_step = step;  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/aboba.txt", "r");  FILE\* data = fopen(  "./experimental\_data/random\_insertion\_time.txt", "w");  double start\_time;  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  start\_time = wtime();  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % tmp\_step == 0) {  fprintf(data, "%d,%.15lf\n", i, (wtime() - start\_time)/i);  tmp\_step += step;  i = 1;  if (tmp\_step > n\_words)  break;  tst\_delete\_tree(tree);  tree = NULL;  start\_time = wtime();  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  tst\_delete\_tree(tree);  }  void sorted\_insertion\_time\_experiment()  {  int tmp\_step = step;  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/sorted\_aboba.txt", "r");  FILE\* data = fopen(  "./experimental\_data/sorted\_insertion\_time.txt", "w");  double start\_time;  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  start\_time = wtime();  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % tmp\_step == 0) {  fprintf(data, "%d,%.15lf\n", i, (wtime() - start\_time)/i);  tmp\_step += step;  i = 1;  if (tmp\_step > n\_words)  break;  tst\_delete\_tree(tree);  tree = NULL;  start\_time = wtime();  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  tst\_delete\_tree(tree);  }  void deleting\_from\_last\_time\_random\_experiment()  {  int tmp\_step = step;  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/aboba.txt", "r");  FILE\* data = fopen(  "./experimental\_data/deleting\_from\_last\_time\_random.txt",  "w");  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  double start\_time;  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % tmp\_step == 0) {  start\_time = wtime();  for (int j = tmp\_step; j > 0; j--)  tree = tst\_delete(tree, strings[j]);  fprintf(data, "%d,%.15lf\n", i, (wtime() - start\_time)/i);  tmp\_step += step;  i = 1;  if (tmp\_step > n\_words)  break;  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  }  void deleting\_from\_last\_time\_sorted\_experiment()  {  int tmp\_step = step;  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/sorted\_aboba.txt", "r");  FILE\* data = fopen(  "./experimental\_data/deleting\_from\_last\_time\_sorted.txt",  "w");  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  double start\_time;  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % tmp\_step == 0) {  start\_time = wtime();  for (int j = tmp\_step; j > 0; j--)  tree = tst\_delete(tree, strings[j]);  fprintf(data, "%d,%.15lf\n", i, (wtime() - start\_time)/i);  tmp\_step += step;  i = 1;  if (tmp\_step > n\_words)  break;  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  }  void random\_lookup\_time\_random\_experiment()  {  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/aboba.txt", "r");  FILE\* data = fopen("./experimental\_data/random\_lookup\_time\_random.txt", "w");  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  double start\_time;  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % step == 0) {  start\_time = wtime();  for (int j = 1; j <= i; j++)  tst\_lookup(tree, strings[j]);  fprintf(data,  "%d,%.10lf\n",  i,  (wtime() - start\_time) / i);  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  tst\_delete\_tree(tree);  }  void random\_lookup\_time\_sorted\_experiment()  {  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/sorted\_aboba.txt", "r");  FILE\* data = fopen("./experimental\_data/random\_lookup\_time\_sorted.txt", "w");  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  double start\_time;  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % step == 0) {  start\_time = wtime();  for (int j = 1; j <= i; j++)  tst\_lookup(tree, strings[j]);  fprintf(data,  "%d,%.10lf\n",  i,  (wtime() - start\_time) / i);  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  tst\_delete\_tree(tree);  }  void last\_lookup\_time\_random\_experiment()  {  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/aboba.txt", "r");  FILE\* data = fopen("./experimental\_data/last\_lookup\_time\_random.txt", "w");  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  double start\_time;  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % step == 0) {  start\_time = wtime();  for (int j = 0; j < n\_iterations; j++)  tst\_lookup(tree, strings[i]);  fprintf(data,  "%d,%.10lf\n",  i,  (wtime() - start\_time) / n\_iterations);  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  tst\_delete\_tree(tree);  }  void last\_lookup\_time\_sorted\_experiment()  {  tst\* tree = NULL;  char\*\* strings = malloc(sizeof(char\*) \* (n\_words + 1));  FILE\* dict = fopen("./data/sorted\_aboba.txt", "r");  FILE\* data = fopen("./experimental\_data/last\_lookup\_time\_sorted.txt", "w");  fprintf(data, "n,t\n");  for (int i = 1; i <= n\_words; i++) {  strings[i] = malloc(11);  fscanf(dict, "%s", strings[i]);  }  double start\_time;  for (int i = 1; i <= n\_words; i++) {  tree = tst\_insert(tree, strings[i]);  if (i % step == 0) {  start\_time = wtime();  for (int j = 0; j < n\_iterations; j++)  tst\_lookup(tree, strings[i]);  fprintf(data,  "%d,%.10lf\n",  i,  (wtime() - start\_time) / n\_iterations);  }  }  for (int i = 1; i <= n\_words; i++)  free(strings[i]);  free(strings);  fclose(dict);  fclose(data);  tst\_delete\_tree(tree);  }  double wtime()  {  struct timeval t;  gettimeofday(&t, NULL);  return (double)t.tv\_sec + (double)t.tv\_usec \* 1E-6;  } |

experimental\_funcs.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 | #pragma once  #include <stdio.h>  #include <sys/time.h>  #include <time.h>  void random\_insertion\_time\_experiment();  void sorted\_insertion\_time\_experiment();  void deleting\_from\_last\_time\_random\_experiment();  void deleting\_from\_last\_time\_sorted\_experiment();  void random\_lookup\_time\_random\_experiment();  void random\_lookup\_time\_sorted\_experiment();  void last\_lookup\_time\_random\_experiment();  void last\_lookup\_time\_sorted\_experiment();  double wtime(); |