МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

СЕКЦІЯ ІКТ

ВИПУСКНА РОБОТА

на тему:

«Порівняльний аналіз реалізації структур даних «Зв'язний список» та «Бінарне дерево пошуку» в прикладних застосуваннях»

Завідувач

випускаючої кафедри **Довбиш А. С.**

Керівник роботи **Шаповалов С. П.**

Студента групи ІНз – 71С **Бершов Є. В.**

СУМИ 2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання
Кафедра комп'ютерних наук

Затверджую___________

Зав. кафедрою Довбиш А.С.

"_____" ____________2021 р.

ЗАВДАННЯ

до випускної роботи

Студента п'ятого курсу, групи ІНз-71С спеціальності "Комп'ютерні науки" заочної форми навчання Бершова Євгенія Володимировича

Тема:«Порівняльний аналіз реалізації структур даних «Зв'язний список» та «Бінарне дерево пошуку» в прикладних застосуваннях»

Затверджена наказом по СумДУ

№________ от ___________2021 р.

Зміст пояснювальної записки: 1) аналітичний огляд застосувань структур даних ; 2) постановка завдання й формування завдань дослідження; 3) опис основних положень, математичних моделей і алгоритмів, що використовуються для рішення поставленого завдання; 5) розробка інформаційного й програмного забезпечення; 6) аналіз результатів моделювання.

Дата видачі завдання "_____"____________2021 р.

Керівник випускної роботи __________ Шаповалов С. П.

Завдання прийняв до виконання ____________ Бершов Є. В.

РЕФЕРАТ

Записка: 51 стор., 15 рис., 7 табл., 1 додаток, 7 джерел інформації.

Об’єкт дослідження — прикладні застосування структур даних.

Мета роботи — комп’ютерний порівняльний аналіз структур даних «Зв’язний список» та «Бінарне дерево пошуку».

Методи дослідження — математичне моделювання, комп’ютерна реалізація алгоритмів на ЕОМ.

Результати — розроблено інформаційне та програмне забезпечення комп’ютерного порівняльного аналізу структур даних «Зв’язний список» та «Бінарне дерево пошуку». Проведено огляд алгоритмів на цих структурах даних. Виконано комп’ютерну реалізацію телефонного довідника за допомогою алгоритмічної мови програмування С++.

ЗВ’ЯЗНИЙ СПИСОК, БІНАРНЕ ДЕРЕВО ПОШУКУ, АЛГОРИТМИ ВСТАВКИ, ПОШУКУ, ВИДАЛЕННЯ, ПОРІВНЯЛЬНИЙ КОМП’ЮТЕРНИЙ АНАЛІЗ,МОВА ПРОГРАМУВАННЯ С++

ЗМІСТ

ВСТУП.. 5

1 АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ 7

1.1Аналітичній огляд застосувань структур даних «Звязний список» та «Бінарне дерево пошуку» 7

1.2 Постановка задачі.. 10

2 ІНФОРМАЦІЙНА ПОСТАНОВКА ЗАВДАННЯ ТА ВИБІР МЕТОДУ ЇЇ РІШЕННЯ. 11

2.1 Опис основних структур даних 11

2.2 Структура «Зв’язний список» та операції 14

2.3 Структура «Бінарне дерево пошуку» та операції 17

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТОВІ РОЗРАХУНКИ……………………………………………………23

3.1 Комп’ютерна реалізація 23

3.2 Тестові розрахунки 25

ВИСНОВКИ.. 34

СПИСОК ЛІТЕРАТУРИ.. 35

ДОДАТОК... 36

ВСТУП

В інформаційно-комунікаційних технологіях – структура даних – це формат даних, управління та зберігання даних, що забезпечує ефективний доступ та модифікацію[1-7]. Більш класичне означення, структура даних – це сукупність значень даних, між якими існує взаємозв'язок та є визначеним набір функцій або операції, які можна застосувати до даних.

Структури даних служать основою для абстрактних типів даних (ADT). ADT визначає логічну форму типу даних. Структура даних реалізує фізичну форму типу даних [1-3].

Різні типи структур даних підходять для різних типів програм, а деякі є вузькоспеціалізованими для конкретних завдань. Наприклад, реляційні бази даних зазвичай використовують індекси B-дерева для отримання даних, тоді як реалізації компілятора зазвичай використовують хеш-таблиці для пошуку ідентифікаторів.

Структури даних використовуються майже в кожній програмі або програмній системі, яка була розроблена. Більше того, структури даних підпадають під основи обчислювальної техніки та програмної інженерії. Це ключова тема, коли мова заходить про питання розробки програмних продуктів. Отже, як розробники, ми повинні добре знати про структури даних.

Структури даних забезпечують засіб ефективного управління великими обсягами даних для використання, таких як великі бази даних та послуги індексування Інтернету. Зазвичай ефективні структури даних є ключовими для розробки ефективних алгоритмів. Деякі формальні методи проектування та мови програмування наголошують на структурі даних, а не на алгоритмах, як на ключовому організуючому факторі проектування програмного забезпечення. Структури даних можна використовувати для організації зберігання та пошуку інформації, що зберігається як в основній, так і в додатковій пам'яті.

Будь-які достатньо великі програми проектуються шляхом декомпозиції задачі – виділення в задачі деяких структур і їхніх абстракцій. Абстрагування від проблеми передбачає ігнорування ряду деталей для того, щоб звести задачу до

більш простої. Задачі абстрагування і наступна декомпозиція типові для процесу створення програм: декомпозиція використовується для поділу програм на компоненти; абстрагування ж передбачає продуманий вибір компонентів для цієї задачі. Структурний підхід до даних і алгоритмів дає можливість структурування складної програми. Розробляти сучасну програмну методом „все відразу" неможливо, вона повинна бути представлена в вигляді деякої структури – складових частин і зв'язків між ними. Правильне структурування дає можливість на кожному етапі розробки зосередити увагу розробника на одній оглядовій її частині або доручити реалізацію різних її частин різним виконавцям

Стають актуальними проведення досліджень, направлених на порівняльний аналіз структур даних на предмет виявлення переваг та недоліків. Основна мета таких досліджень – виявлення сфери застосовності тих, чи інших структур.

ADT розповідає, що потрібно робити, а структура даних - як це робити. Іншими словами, ми можемо сказати, що ADT дає нам схему, тоді як структура даних забезпечує частину реалізації. Тепер виникає питання: як можна дізнатися, яку структуру даних використовувати для конкретного ADT ?

Для надання відповіді на це питання потрібно провести порівняльний аналіз конкретних застосувань структур даних й надати потрібні рекомендації.

Оскільки різні структури даних можуть бути реалізовані в конкретному ADT, але різні реалізації порівнюються за часом і простором. Наприклад, стек ADT може бути реалізований як масивами, так і пов'язаним списком. Припустимо, що масив забезпечує часову ефективність, тоді як зв'язаний список забезпечує космічну ефективність, тому буде обраний той, який найкраще відповідає поточним вимогам. Враховуючи нове розуміння, нові методики та потужну підтримку апаратного забезпечення, ми вважаємо будуть поширюватися дослідження саме порівняльного характеру.

1 АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналітичній огляд застосувань структур даних «Зв'язний список» та «Бінарне дерево пошуку»

Структури даних є одними з основних та базових елементів інформаційно-комуніувційних технологій. Структура даних - це спеціалізований формат для організації, обробки, отримання та зберігання даних. Структури даних полегшують користувачам доступ до потрібних даних та роботу з ними відповідними способами. Найголовніше, що структури даних визначають організацію інформації, щоб машини та люди могли краще її зрозуміти.

В інформатиці та комп'ютерному програмуванні може бути обрана або призначена структура даних для зберігання даних з метою їх використання з різними алгоритмами. У деяких випадках основні операції алгоритму тісно пов'язані з дизайном структури даних. Кожна структура даних містить інформацію про значення даних, взаємозв'язки між даними та - в деяких випадках - функції, які можна застосувати до даних.

Структури даних часто класифікують за їх характеристиками [1-7]:

- Лінійний або нелінійний. Ця характеристика описує, розташовані елементи даних у послідовному порядку, наприклад, з масивом, або в невпорядкованій послідовності, наприклад, з графіком.
- Однорідні або неоднорідні. Ця характеристика описує, чи всі елементи даних у даному сховищі є однотипними. Одним із прикладів є колекція елементів у масиві або різних типів, таких як абстрактний тип даних, визначений як структура в C або специфікація класу в Java.
- Статичний або динамічний. Ця характеристика описує спосіб компіляції структур даних. Статичні структури даних мають фіксовані розміри, структури та розташування пам'яті під час компіляції. Динамічні структури даних мають розміри, структури та місця пам'яті, які можуть зменшуватися або розширюватися залежно від використання

Структури даних логічно поєднують елементи даних та сприяють ефективному використанню, збереженню та обміну даними. Вони забезпечують офіційну модель, яка описує спосіб організації елементів даних

Важливо не тільки використовувати структури даних, але також важливо правильно вибрати структуру даних для кожного завдання. Вибір неправильно підібраної структури даних може призвести до повільного виконання або неактивного коду. П'ять факторів, які слід враховувати при виборі структури даних, включають наступне:

- ✓ Яка інформація буде зберігатися?
- ✓ Як буде використана ця інформація?
- ✓ Де слід зберігати або зберігати дані після їх створення?
- ✓ Який найкращий спосіб упорядкувати дані?
- ✓ Які аспекти управління пам'яттю та резервуванням зберігання слід враховувати?

Як використовуються структури даних?

Загалом, структури даних використовуються для реалізації фізичних форм абстрактних типів даних. Структури даних є найважливішою частиною проектування ефективного програмного забезпечення. Вони також відіграють важливу роль у розробці алгоритмів та в тому, як ці алгоритми використовуються в комп'ютерних програмах.

При комп'ютерному програмуванні може бути обрана або призначена структура даних для зберігання даних з метою роботи над нею з різними алгоритмами. Кожна структура даних містить інформацію про значення даних, взаємозв'язки між даними та функції, які можна застосувати до даних.

Стає питання чим користатися при обиранні тієї чи іншої структури даних при виконанні конкретного завдання?

Тут на допомогу може прийти накоплений досвід з використання загальновідомих структур даних [1-7]. Використаємо його при виборі структур даних для виконання нашого завдання.

Зв'язані списки найкраще обирати, якщо програма керує колекцією предметів, які не потрібно впорядковувати, для додавання чи вилучення елемента з колекції потрібен постійний час і збільшення часу пошуку є нормальним.

Ось перелік основних задач для використання структури даних «Зв'язаний список»:

- Реалізація стеків, черг, двійкових дерев та графіків заздалегідь визначеного розміру.
- Впровадити функції динамічного управління пам'яттю операційної системи.
- Круговий зв'язаний список використовується у слайд-шоу, де користувач хоче повернутися до першого слайда після відображення останнього слайда.
- Кругова черга використовується для підтримки послідовності гри кількох гравців у грі.
- Зображення пов'язані між собою. Отже, програмне забезпечення для перегляду зображень використовує зв'язаний список для перегляду попереднього та наступного зображень за допомогою попередньої та наступної кнопок.
- Доступ до веб-сторінок можна отримати за допомогою попереднього та наступного посилань на URL-адреси, які пов'язані за допомогою пов'язаного списку.

Бінарні дерева добре підходять для управління колекцією предметів, що мають стосунки батьків та дітей, наприклад, сімейним деревом. Бінарні дерева пошуку підходять для управління відсортованою колекцією, де метою є оптимізація часу, необхідного для пошуку конкретних елементів у колекції.

Ось перелік основних задач для використання структури даних «Бінарне дерево пошуку»:

- Впровадження ієрархічних структур в комп'ютерних системах, таких як каталог та файлова система.
- Генерація коду, наприклад, код Хаффмана.

- Прийняття рішень в ігрових додатках.
- Розбір виразів та висловлювань у компіляторах мови програмування
- Для зберігання ключів даних для СУБД для індексації
- Об’ємні дерева для прийняття рішень щодо маршрутизації в комп’ютерних та комунікаційних мережах
- Хеш дерева
- алгоритм пошуку шляхів для реалізації в ШІ, робототехніці та додатках для відеоігор
- XML-аналізатор використовує алгоритми дерева.
- Алгоритм, заснований на прийнятті рішень, використовується в машинному навчанні, яке працює на алгоритмі дерева.
- Бази даних також використовують деревні структури даних для індексації.
- Сервер доменних імен (DNS) також використовує деревоподібні структури.
- Провідник файлів / мій мобільний комп’ютер / будь-який комп’ютер
- BST використовується в комп'ютерній графіці.
-

1.2 Постановка задачі

Поставимо наступне завдання для дослідження.

Розробити комп’ютерну програму, яка буде відігравати роль телефонного довідника. Для реалізації використати два типи структур даних: «Зв’язний список» та «Бінарне дерево пошуку».

Для досягнення мети знайти рішення підзадач:

1. *Дослідити основні операції та порівняти їх використання.*
2. *Провести порівняльний аналіз цих алгоритмів на тестових завданнях.*
3. *На основі проведених досліджень зробити висновки застосовності структур даних.*

2 ІНФОРМАЦІЙНА ПОСТАНОВКА ЗАВДАННЯ ТА ВИБІР МЕТОДУ ЇЇ РІШЕННЯ

2.1 Опис основних структур даних

Структура даних - це спосіб зберігати та впорядковувати дані для ефективного їх використання. Структура даних - це не будь-яка мова програмування, така як C, C ++, java тощо. Це набір алгоритмів, які ми можемо використовувати в будь-якій мові програмування для структурування даних у пам'яті.

Наступні терміні є основними умовами структури даних.

Інтерфейс - кожна структура даних має інтерфейс. Інтерфейс представляє набір операцій, які підтримує структура даних. Інтерфейс надає лише перелік підтримуваних операцій, тип параметрів, які вони можуть прийняти, і тип повернення цих операцій.

Впровадження. Впровадження забезпечує внутрішнє представлення структури даних. Реалізація також забезпечує визначення алгоритмів, що використовуються в операціях структури даних.

Характеристики структури даних.

Правильність - реалізація структури даних повинна правильно реалізовувати свій інтерфейс.

Складність у часі - час роботи або час виконання операцій зі структурою даних повинен бути якомога меншим.

Складність простору - використання пам’яті операцією структури даних повинно бути якомога меншим.

Випадки часу виконання.

Є три випадки, які зазвичай використовуються для порівняння часу виконання різної структури даних відносно.

1. Найгірший випадок - це сценарій, коли певна операція зі структурою даних займає максимум часу, який вона може зайняти. Якщо найгіршим часом операції є f (n), то ця операція не займе більше f (n) часу, де f (n) представляє функцію n.

2. Середній випадок - це сценарій, що відображає середній час виконання операції зі структурою даних. Якщо на виконання операції потрібно f (n) часу, то m операцій займе mf (n) часу.
3. Найкращий випадок - це сценарій, що відображає найменший можливий час виконання операції зі структурою даних. Якщо на виконання операції потрібно f (n) часу, то фактична операція може зайняти час як випадкове число, яке було б максимальним як f (n).

Переваги структур даних.

Нижче наведено переваги структури даних:

Ефективність: Якщо вибір структури даних для реалізації певного ADT правильний, це робить програму дуже ефективною з точки зору часу та простору.

Багаторазове використання: структури даних забезпечують багаторазове використання, що означає, що кілька клієнтських програм можуть використовувати структуру даних.

Абстракція: Структура даних, визначена ADT, також забезпечує рівень абстракції. Клієнт не може бачити внутрішню роботу структури даних, тому йому не доведеться турбуватися про частину реалізації. Клієнт може бачити лише інтерфейс.

Основні операції.

Дані в структурах даних обробляються певними операціями. Вибрана конкретна структура даних багато в чому залежить від частоти операції, яку потрібно виконати над структурою даних.

Основні операції наступні: Обхід, Пошук, Вставка, Видалення, Сортування, Злиття.

Потреба в структурі даних.

Оскільки програми ускладнюються, а дані збагачуються, існує три найпоширеніші проблеми, з якими зараз стикаються програми.

Пошук даних - розгляньте опис 1 мільйона (106) предметів магазину. Якщо програма шукає елемент, вона повинна шукати елемент у 1 мільйоні (106) елементів кожного разу, сповільнюючи пошук. У міру зростання даних пошук стане повільнішим.

Швидкість процесора - хоча і дуже висока швидкість процесора, вона обмежується, якщо дані зростають до мільярда записів.

Кілька запитів - Оскільки тисячі користувачів можуть одночасно шукати дані на веб-сервері, навіть швидкий сервер не працює під час пошуку даних.

Для вирішення вищезазначених проблем на допомогу приходять структури даних. Дані можуть бути організовані в структурі даних таким чином, що пошук усіх елементів може не вимагати, а необхідні дані можна шукати майже миттєво.

Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх обробки. Добре побудовані структури даних дозволяють оптимізувати використання машинного часу та пам'яті комп'ютера для виконання найкритичніших операцій.

Структури даних, як правило, засновані на здатності комп'ютера отримувати та зберігати дані в будь-якому місці його пам'яті, заданому покажчиком - бітовим рядком, що представляє адресу пам'яті, який може сам зберігатися в пам'яті та маніпулювати програмою. Таким чином, структури даних масиву та запису засновані на обчисленні адрес елементів даних за допомогою арифметичних операцій, тоді як зв’язані структури даних засновані на збереженні адрес елементів даних у самій структурі.

Реалізація структури даних зазвичай вимагає написання набору процедур, які створюють та маніпулюють екземплярами цієї структури. Ефективність структури даних не може бути проаналізована окремо від цих операцій. Це спостереження мотивує теоретичну концепцію абстрактного типу даних, структуру даних, яка опосередковано визначається операціями, які можуть бути виконані з ними, та математичними властивостями цих операцій (включаючи їх простір та час) [1-7].

2.2 Структура «Зв’язний список» та операції

Зв’язаний список - це лінійна структура даних, в якій елементи не зберігаються у суміжних місцях пам'яті. Елементи у зв’язаному списку пов’язані за допомогою покажчиків, як показано на рис. 2.2: Зв’язний список можна візуалізувати як ланцюжок вузлів, де кожний вузол вказує на наступний вузол.

Рисунок 2.1 – Візуалізація зв’язного списку

Іншими словами, зв’язний список складається з вузлів, де кожен вузол містить поле даних та посилання (посилання) на наступний вузол у списку.
Переваги над масивами - 1) Динамічний розмір;2) Простота вставки / видалення
Недоліки:
1) Випадковий доступ не дозволяється. Ми повинні отримувати доступ до елементів послідовно, починаючи з першого вузла. Отже, ми не можемо ефективно виконувати двійковий пошук зі зв’язаними списками з його реалізацією за замовчуванням. Прочитайте про це тут.
2) Для кожного елемента списку потрібен додатковий простір пам'яті для вказівника.
3) Не зручно кешувати. Оскільки елементи масиву є суміжними розташуваннями, існує місцевість посилання, якої немає у випадку пов’язаних списків.

Зв'язний список – базова динамічна структура даних в інформаційно-комунікаційних технологіях, що складається з вузлів, кожен з яких містить як власне дані, так і одну або дві посилання («зв'язки») на наступний і / або попередній вузол списку[1-3].

Зв’язані списки є одними з найпростіших і найпоширеніших структур даних. Вони можуть бути використані для реалізації кількох інших загальноприйнятих абстрактних типів даних, включаючи списки, стеки, черги, асоціативні масиви та S-вирази, хоча нерідкі випадки, коли ці структури даних реалізовуються безпосередньо без використання пов'язаного списку як основи.

Розглянемо основні операції.

1.Операція вставки.

Додавання нового вузла у зв'язаний список - це більш ніж один крок діяльності. Про це ми дізнаємось із діаграм тут. Спочатку створіть вузол, використовуючи ту саму структуру, і знайдіть місце, куди його потрібно вставити.

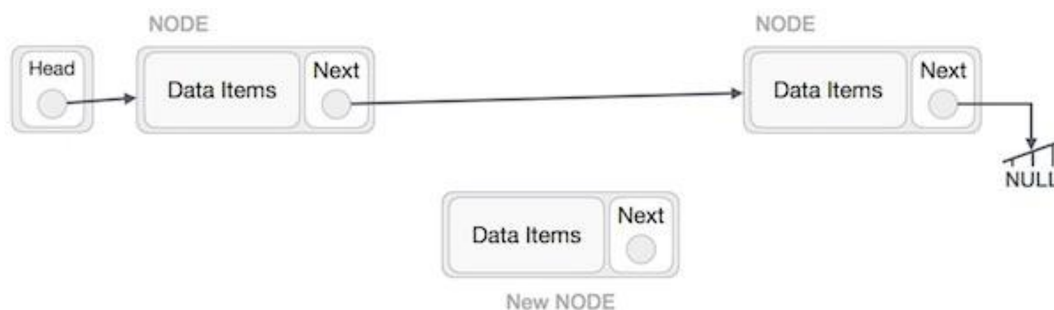

Рисунок 2.2 – Ілюстрація до операції вставка

Уявіть, що ми вставляємо вузол B (NewNode), між A (LeftNode) і C (RightNode). Тоді точка B.поруч з C –

```
NewNode.next -> RightNode;
```

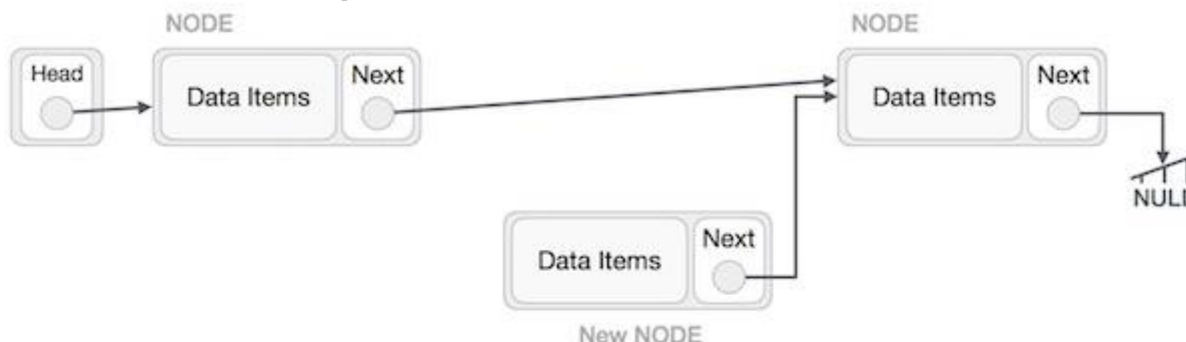

Рисунок 2.3 – Ілюстрація до операції вставка

Тепер наступний вузол зліва повинен вказувати на новий вузол.

```
LeftNode.next -> NewNode;
```

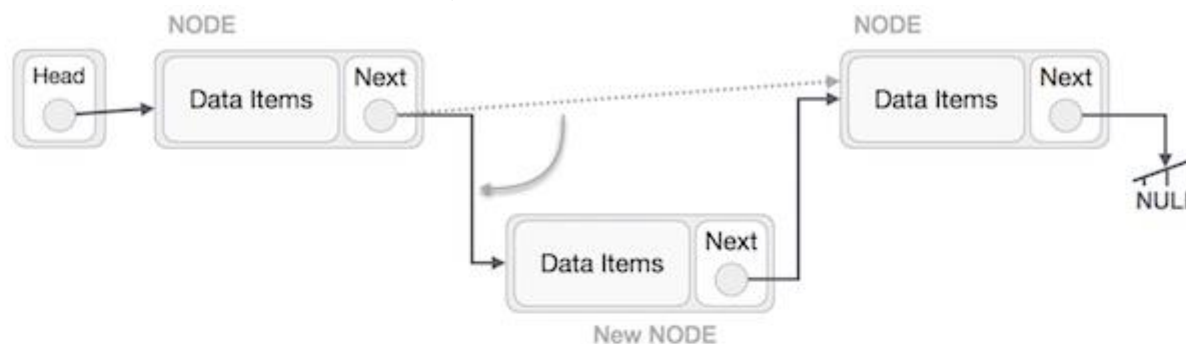

Рисунок 2.4 – Ілюстрація до операції вставка

Це поставить новий вузол посередині двох. Новий список повинен виглядати так –

Рисунок 2.5 – Операція вставка завершена

Подібні кроки слід робити, якщо вузол вставляється на початку списку. Вставляючи його в кінці, другий останній вузол списку повинен вказувати на новий вузол, а новий вузол вказуватиме на NULL.

.. Наступний код демонструє операцію вставки на початку подвійно пов'язаного списку.

```
//insert link at the first location
void insertFirst(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //update first prev link
      head->prev = link;
   }

   //point it to old first link
   link->next = head;

   //point first to new first link
   head = link;
}
```

1. *Операція видалення*

Наступний код демонструє операцію видалення на початку подвійно пов'язаного списку.

```
//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;
```

```
   //if only one link
   if(head->next == NULL) {
      last = NULL;
   } else {
      head->next->prev = NULL;
   }

   head = head->next;

   //return the deleted link
   return tempLink;
}
```

2. *Вставка в кінець списку*

Наступний код демонструє операцію вставки в останню позицію подвійно пов'язаного списку.

```
//insert link at the last location
void insertLast(int key, int data) {

   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

   if(isEmpty()) {
      //make it the last link
      last = link;
   } else {
      //make link a new last link
      last->next = link;

      //mark old last node as prev of new link
      link->prev = last;
   }

   //point last to new last node
   last = link;
}
```

2.3 Структура «Бінарне дерево пошуку» та операції

В інформаційно-комунікаційних технологіях окрім структур даних що несуть в собі лінійний характер (наприклад, «Зв'язний список»), частіше застосовуються нелінійні або розгалужені структури [1-5]. Яскравим представником останніх є дерева, що входять до поняття графа. Їх виділяє те, що вони мають задають порядок в розташуванні зберігаємої інформації завдяки своїм властиво-

стям. Бінарні дерева пошуку - це фундаментальна структура даних, яка використовується для побудови більш абстрактних структур даних, таких як набори, мультимножини та асоціативні масиви.

Бінарне дерево пошуку (BST), як випливає з назви, - це бінарне дерево, де дані організовані в ієрархічну структуру. Ця структура даних зберігає інформацію у відсортованому порядку.

Кожен вузол у двійковому дереві пошуку містить такі атрибути.

- ключ: Значення, що зберігається у вузлі.
- ліворуч: вказівник на ліву дитину.
- праворуч: вказівник на потрібну дитину.
- p: Вказівник на батьківський вузол.

Бінарне дерево пошуку має унікальну властивість, що відрізняє його від інших дерев. Ця властивість відома як властивість бінарного дерева пошуку.

Нехай x - вузол у двійковому дереві пошуку.

Якщо y - вузол у лівому піддереві x, тоді $y.key \leq x.key$

Якщо y - вузол у правому піддереві x, тоді $y.key \geq x.key$

Часто інформація, представлена кожним вузлом, є записом, а не окремим елементом даних. Однак для цілей послідовності вузли порівнюються за їх ключами, а не за якою-небудь частиною пов'язаних із ними записів.

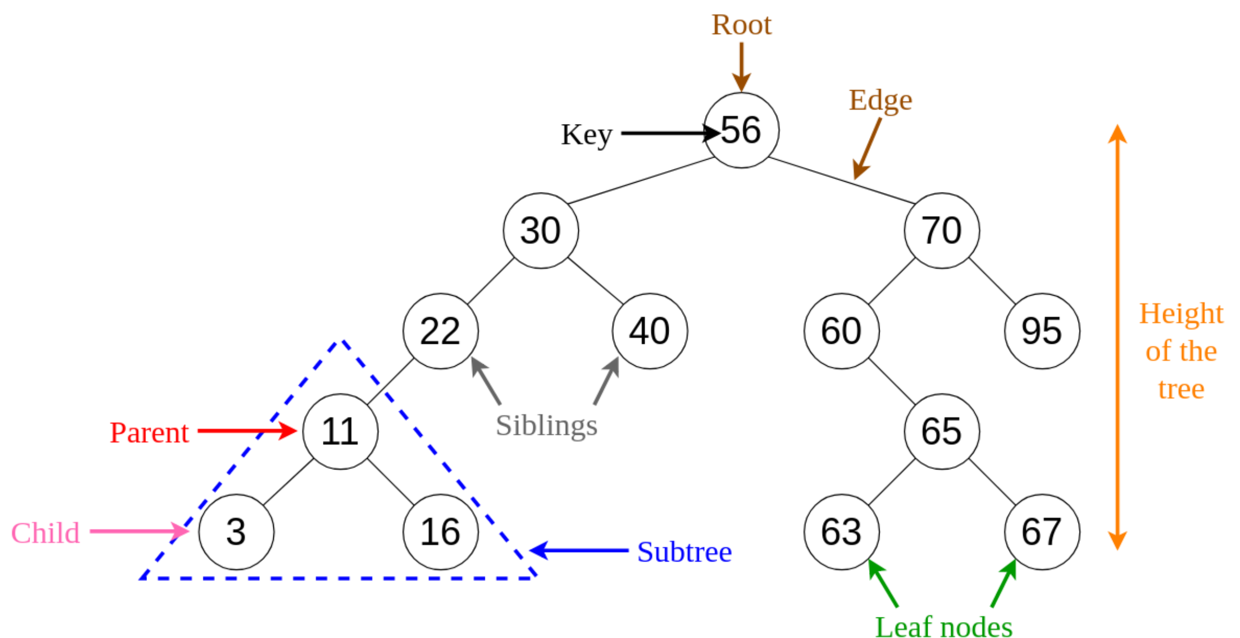

Рисунок 2.6 – Бінарне дерево пошуку з атрибутами

Бінарні дерева пошуку підтримують три основних операції: вставка елементів, видалення елементів та пошук елементів (перевірка наявності ключа).

1. Пошук елементів.

Пошук у двійковому дереві пошуку певного ключа може бути запрограмований рекурсивно або ітеративно.

Для пошуку вузла із заданим ключем в бінарному дереві пошуку використовується наступна процедура Tree_Search, яка отримує як параметри покажчик на корінь бінарного дерева і ключ k, а повертає покажчик на вузол з цим ключем (якщо такий існує; в іншому випадку повертається значення NIL)

Алгоритм пошуку нескладний й може бути описаний наступне.

1. Почнемо пошук з кореневого вузла. Якщо дерево нульове, ключ, який ми шукаємо, у дереві не існує.

2. В іншому випадку, якщо ключ дорівнює ключу кореня, пошук успішний, і ми повертаємо вузол.

3. Якщо ключ менше, ніж у кореня, ми шукаємо ліве піддерево. Подібним чином, якщо ключ більший, ніж ключ кореня, ми шукаємо потрібне піддерево.

4. Цей процес повторюється доти, доки ключ не буде знайдений або решта піддерева буде нульовим. Якщо шуканий ключ не знайдений після досягнення нульового піддерева, тоді ключа немає в дереві.

Псевдокодом цей алгоритм має вид

Tree_Search (x, k)
1. if x = nil або k = key [x] 2.   then return x 3. if k < key [x] 4.   then return Tree_Search (left [x], k)

5. else return Tree_Search (right [x], k)

Оскільки в гіршому випадку цей алгоритм повинен здійснювати пошук від кореня дерева до найдальшого від кореня листка, операція пошуку займає час, пропорційний висоті дерева.

2. Вставка нового вузла.

Для вставки нового значення v в бінарне дерево пошуку T ми скористаємося процедурою TREE_INSERT. Процедура отримує як параметр вузол z, у якого key[z] = v, left[z] = NIL і right[z] = NIL, після чого вона таким чином змінює T і деякі поля z, що z виявляється вставленим в відповідну позицію в дереві.

Алгоритм вставки представим наступне.

Вставка починається так, як би розпочався пошук; якщо ключ не дорівнює ключу кореня, ми шукаємо ліве або праве піддерева, як і раніше. Зрештою, ми досягнемо зовнішнього вузла і додамо нову пару ключ-значення (тут закодовану як запис 'newNode') як її праву або ліву дочірню структуру, залежно від ключа вузла. Іншими словами, ми перевіряємо корінь і рекурсивно вставляємо новий вузол у ліве піддерево, якщо його ключ менше, ніж у кореня, або праве піддерево, якщо його ключ більший або дорівнює кореневому.

Псевдокод алгоритму вставки

TREE_INSERT (T, Z)
1. y ← NIL 2. x ← root[T] 3. while x ≠ NIL 4. do y ← x 5. if key[z] < key [x] 6. then x ← left[x]

```
7.                              else x ← right[x]
8. p [z] ← y
9. if y = NIL
10.       then root[T] ← z                         // Дерево T - пусте
11.       else if key[z] <key[y]
12.                 then left[y] ← z
13.                 else right[y] ← z
```

3. *Операція видалення елементу.*

Видаляючи вузол із бінарного дерева пошуку, обов’язково слід підтримувати послідовність послідовності вузлів у порядку. Є багато можливостей зробити це. Однак наступний метод, запропонований Т. Гіббардом у 1962 р. [1-3], гарантує, що висоти предметних піддерев змінюються щонайбільше на одну. Є три можливі випадки для розгляду:

1.Видалення вузла без дітей: просто видаліть вузол з дерева.

2.Видалення вузла з однією дочірньою системою: видаліть вузол і замініть його дочірнім.

3.Видалення вузла D з двома дітьми: виберіть або попередника D, або послідовника E (див. рис. 2.6). Замість того, щоб видаляти D, перепишіть його ключ і значення буквою E. Якщо E не має дочірнього вузла, видаліть E з попереднього батьківського G. Якщо E має дочірній F, це правильний дочірній, так що це замінити E на батьківський.

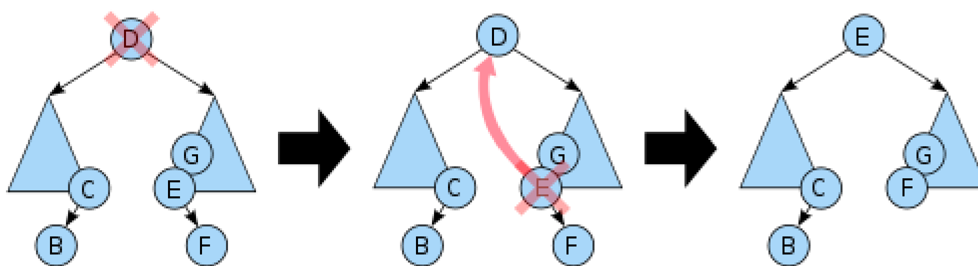

Рисунок 2.7 – Операція видалення вузла

У всіх випадках, коли D виявляється коренем, виконайте заміну кореневого вузла знову.

Вузли з двома дітьми важче видалити. Наступник вузла в порядку - це найлівіший дочірній елемент його правого піддерева, а попередник вузла в порядку - самий правий дочірній елемент лівого піддерева. В обох випадках цей вузол матиме лише одну дитину або її взагалі не буде. Видаліть його відповідно до одного з двох простих випадків, наведених вище.

Послідовне використання послідовника в порядку або попередника в порядку для кожного екземпляра випадку двох дітей може призвести до незбалансованого дерева, тому деякі реалізації вибирають те чи інше в різний час.

Аналіз часу роботи: Хоча ця операція не завжди пересуває дерево до листя, це завжди є можливою; тому в гіршому випадку потрібен час, пропорційний висоті дерева. Це не вимагає більшої кількості, навіть якщо вузол має двох дочірніх організацій, оскільки він все ще йде по одному шляху і не відвідує жодного вузла двічі.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТОВІ РОЗРАХУНКИ

3.1 Комп'ютерна реалізація

В якості мови програмування для реалізації проекту обрано С++, як одну з найбільш часто використовуваних мов в світі, що має широкий спектр компіляторів, які працюють на різних платформах. Як середовище розробки під Windows було використовувано Visual Studio 2008.

Microsoft Visual Studio - це інтегроване середовище розробки (IDE) від Microsoft. Застосовується для розробки комп'ютерних програм, а також веб-сайтів, веб-програм, веб-сервісів та мобільних додатків. Visual Studio використовує платформи розробки програмного забезпечення Microsoft, такі як Windows API, Windows Forms, Windows Presentation Foundation, Windows Store та Microsoft Silverlight. Він може створювати як власний код, так і керований код.

С ++ підтримує процедурну, узагальнену, і об'єктно-орієнтовану парадигми програмування, і багато інших парадигм. Код стандартної бібліотеки С ++ буде працювати на багатьох платформах. Все це надає С++ привабливості коли обирається мова програмування для реалізації проекту.

Після підключення стандартних бібліотек

```
#include <iostream>
#include <fstream>
#include <string>
```

програма визиває меню

```
void menu(List l, Tree t) {
	cout << "Choose operation" << endl;
	cout << "Enter \"Add\" to add new element, \"Find\" to get info.\nEnter \"Inf
o\" to show all information, \"Delete\" to delete element, \"Exit\" to end program"
<< endl;
	string op;
	cin >> op;
	if (op == "Add") {
```

```
		cout << endl;

	cout << "Enter \"List\" to add to list. Enter \"Tree\" to add to tree. Enter \"Bot
h\" to add to both structures" << endl;
		cin >> op;
		if (op == "List") {
			string name, phone;
			cout << "Enter name: ";
			cin >> name;
			cout << "Enter phone: ";
			cin >> phone;
			l.addHead(name, phone);
			cout << endl;
		}
		else if (op == "Tree") {
			string name, phone;
			cout << "Enter name: ";
			cin >> name;
			cout << "Enter phone: ";
			cin >> phone;
			t.insert(name, phone);
		}
```

Так як, за постановкою задачі для складання телефонного довідника використовуються дві структури даних – «зв’язний список» та «бінарне дерево», то дані одночасно записуються і в список і бінарне дерево пошуку.

Ці структури мають свої описання

```
struct nodeL {
	string name;
	string phone;
```

```
	nodeL* next;
	nodeL* prev;
};
```

для class List

та

```
struct nodeT {
	string name;
	string phone;
	nodeT* left;
	nodeT* right;
	nodeT* parent;
};
```

для class Tree .

Далі користувачеві пропонується обрати операцію: додати елемент, знайти телефон за прізвищем, видалити елемент, вивести весь довідник, закінчити роботу з програмою:

```
void insert(string name, string phone) ;
void deleteName(string name) :
void deleteNode(int pos = 0) ;
```

Потім, в залежності від операції, обирається структура даних для роботи – дерево, список або ж обидві одразу. Після виконання операції меню запускається знову

```
void menu(List l, Tree t) .
```

3.2 Тестові розрахунки

Приклад №1. Незначна кількість клієнтів (10).

Тестування розпочнемо з випадку, коли телефонний довідник має незначні вхідні дані. Протестуємо процедуру видалення. Перевіримо видалення клієнта,

який буде коренем для бінарного дерева пошуку і хвостом двозв'язного списку. Такий випадок є найбільш складним при видаленні.

Потім перевіримо видалення клієнта, який буде головою списку і випадковим вузлом для дерева. І на останнє перевіримо видалення клієнта з середини списку та дерева.

Після визначення працездатності програми, відмічаємо час виконання процедури видалення для аналізу. Паралельно також визначаємо кількість операцій порівняння та присвоєння, проведених при виконанні алгоритмів.

Кількість операцій в алгоритмах визначаються за допомогою вбудованих до функцій лічильників.

Зауваження. Якщо операції вставки та видалення приходяться на початок списку, то вони відбудуться миттєво. Якщо ж обрати інший елемент списку – операції відрізнятимуться.

Розглянемо невеликий телефонний довідник. Вхідний файл – book.txt. Він складається із 10-ти пар імен і номерів. Його зміст:

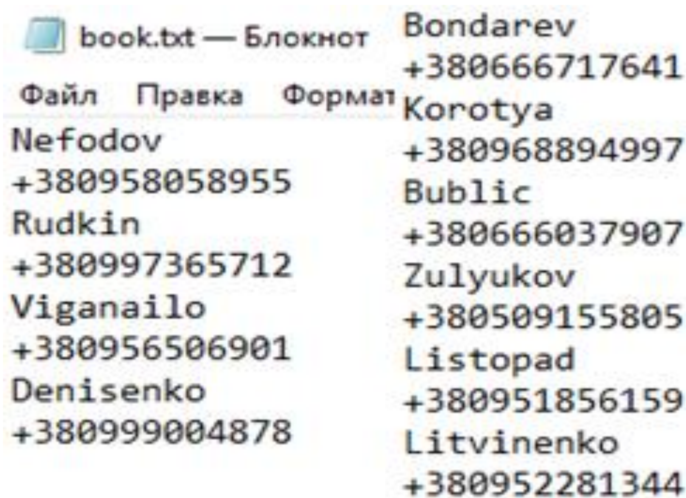
book.txt — Блокнот
Файл Правка Формат
Nefodov
+380958058955
Rudkin
+380997365712
Viganailo
+380956506901
Denisenko
+380999004878
Bondarev
+380666717641
Korotya
+380968894997
Bublic
+380666037907
Zulyukov
+380509155805
Listopad
+380951856159
Litvinenko
+380952281344

Рисунок 3. 1 – Вхідні дані телефонного довідника

Перейдемо до виконання комп'ютерної реалізації.

Нижче приведений скріншот телефонного довідника. Окремо виведено реалізацію на структурі даних «звязний список» та «бінарне дерево пошуку».

```
Phone book in list
Name: Litvinenko
Phone: +380952281344

Name: Listopad
Phone: +380951856159

Name: Zulyukov
Phone: +380509155805

Name: Bublic
Phone: +380666037907

Name: Korotya
Phone: +380968894997

Name: Bondarev
Phone: +380666717641

Name: Denisenko
Phone: +380999004878

Name: Viganailo
Phone: +380956506901

Name: Rudkin
Phone: +380997365712

Name: Nefodov
Phone: +380958058955
```

```
Phone book in tree
Name: Bondarev
Phone: +380666717641

Name: Bublic
Phone: +380666037907

Name: Denisenko
Phone: +380999004878

Name: Korotya
Phone: +380968894997

Name: Listopad
Phone: +380951856159

Name: Litvinenko
Phone: +380952281344

Name: Nefodov
Phone: +380958058955

Name: Rudkin
Phone: +380997365712

Name: Viganailo
Phone: +380956506901

Name: Zulyukov
Phone: +380509155805
```

Рисунок 3. 2 – Реалізація на структурах даних «Зв’язний список» та «бінарне дерево пошуку»

Тепер введемо 3 прізвища Для видалення зі списку– Nefodov – початковий у файлі book.txt, Litvinenko – кінцевий у файлі, Bondarev – із середини файлу. Далі скріншот виведення списку і дерева після видалення.

```
Phone book in list after deleting
Name: Listopad
Phone: +380951856159

Name: Zulyukov
Phone: +380509155805

Name: Bublic
Phone: +380666037907

Name: Korotya
Phone: +380968894997

Name: Denisenko
Phone: +380999004878

Name: Viganailo
Phone: +380956506901

Name: Rudkin
Phone: +380997365712
```

```
Phone book in tree after deleting
Name: Bublic
Phone: +380666037907

Name: Denisenko
Phone: +380999004878

Name: Korotya
Phone: +380968894997

Name: Listopad
Phone: +380951856159

Name: Rudkin
Phone: +380997365712

Name: Viganailo
Phone: +380956506901

Name: Zulyukov
Phone: +380509155805
```

Рисунок 3. 3 – Дані після виконання операції видалення

Як бачимо, виведено по 7 елементів, стільки і має бути. Отже дерево спокійно може видаляти корінь та будь-які інші елементи, а список – позбуватися голови чи хвоста.

Тепер вирахуємо час додавання вузла, пошуку та видалення через спеціальну бібліотеку. Результати операцій за часом.

```
Time of insert in list: 1.86784e-08
Time of insert in tree: 3.3206e-08

Time of search in list: 4.15075e-09
Time of search in tree: 1.86784e-08

Time of deleting in list: 8.3015e-09
Time of deleting in tree: 2.17914e-08
```

Рисунок 3. 4 – Виведення на екран часу виконання операцій при n=10

Залишилося знайти кількість операцій. Виведемо результати у консоль.

```
List. Adding element
Compare operations: 2
Assign operations: 7

Tree. Adding element
Compare operations: 5
Assign operations: 15
```

```
List. Search of element
Compare operations: 8
Assign operations: 8

Tree. Search of element
Compare operations: 3
Assign operations: 2
```

```
List. Deleting element
Compare operations: 13
Assign operations: 16

Tree. Deleting element
Compare operations: 7
Assign operations: 5
```

Рисунок 3. 5 – Виведення на екран кількості операцій присвоєнь та порівнянь

Приклад № 2. Значна кількість клієнтів (1000).

Знайдемо тепер час операцій для більшого об’єму даних. Програма та сама, тільки вводитимемо інший файл із більшою кількістю різної інформації.

```
Time of insert in list: 6.10915e-07
Time of insert in tree: 1.93195e-07

Time of search in list: 3.55062e-06
Time of search in tree: 8.35439e-07

Time of deleting in list: 6.73573e-06
Time of deleting in tree: 1.25316e-06
```

Рисунок 3. 6 – Виведення на екран часу виконання операцій при n=1000

Також знайдемо кількість операцій через лічильники.

```
List. Adding element          List. Search of element     List. Deleting element
Compare operations: 2         Compare operations: 48      Compare operations: 158
Assign operations: 7          Assign operations: 48       Assign operations: 161

Tree. Adding element          Tree. Search of element     Tree. Deleting element
Compare operations: 19        Compare operations: 31      Compare operations: 21
Assign operations: 43         Assign operations: 16       Assign operations: 12
```

Рисунок 3. 7 – Виведення на екран кількості операцій присвоєнь та порівнянь

Декілька слів про порівняльний аналіз.

Тестування проводилися на комп'ютері з наступною конфігурацією:

- ПРОЦЕССОР Intel Core i3 3240 CPU@3.40 GHz;
- ОПЕРАТИВНА ПАМ'ЯТЬ (ОРЕ) 4.0 ГБ (3.11 ГБ досяжно);
- ОПЕРАЦІЙНА СИСТЕМА Windows 10.

При використанні структур даних «Зв’язний список» та «бінарне дерево пошуку» телефонні довідники заповнювалися однаковими базами даних інформації. Порівняльний аналіз операцій на структурах проходив теж ідентично. Тобто операція пошуку обох структур шукала однакове прізвище. А операція видалення видаляла інше прізвище, але також однакове для списку та дерева.

Знайдемо об’єм оперативної пам’яті, що використовувалась структурами через стандартну функцію sizeof(). Вона повертає у байтах кількість займаної пам’яті. Якщо ж аргументом sizeof є посилання, ми отримаємо розмір пов'язаного з нею об'єкта.

```
Size of list: 64
Size of tree: 68
```

Рисунок 3. 8 – Виведення скільки займає пам’яті телефонний довідник на структурі даних «Зв’язний список» та «Бінарне дерево» відповідно

На рис. 3.8 приведено кількість байтів, що виділяється ЕОМ під один і той же телефонний довідник, реалізований на обраних структурах даних відповідного типу. Дерево займає трохи більше об’єм пам’яті, адже має додатковий покажчик на батьківський вузол.

Порівняння часу виконання операцій на структурах даних.

Надалі результати порівняльного аналізу будемо представляти в табличному ракурсі. В табл. 3.1 наведені дані по часу виконання основних операцій на структурах даних при різній кількості вхідних даних. Позначення в таблиці – «нс» – наносекунди, «мкс» – мікросекунди.

Таблиця 3.1 Час виконання основних операцій в телефонному довіднику, побудованого на різних структурах даних

	10 вузлів		1000 вузлів	
	Список	Дерево	Список	Дерево
Вставка	18,68 нс	33,21 нс	0,61 мкс	0,19 мкс
Пошук	4,15 нс	18,68 нс	3,6 мкс	0,84 мкс
Видалення	8,3 нс	21,79 нс	6,74 мкс	1,25 мкс

У таблиці зеленим кольором позначено кращий час. Із таблиці можна помітити, що при малій кількості вузлів дерево програє списку. Це може бути спричинено складністю алгоритмів бінарного дерева пошуку. Попри все, при збільшенні кількості вхідних даних дерево починає випереджувати список у всіх операціях.. У будь-якому випадку, час роботи достатньо малий, тому можлива похибка часу. Адже такий малий час важко вимірюється і на операції початку замірювання часу та кінця також витрачається певний час, який може впливати на результат.

Таблиця 3.2 Об’єм оперативного пам’яті, необхідний для зберігання телефонного довідника, побудованого на різних структурах даних

Використання пам'яті					
10 елементів			1000 елементів		
Список	Дерево	Різниця	Список	Дерево	Різниця
640 б	680 б	40 б	62,5 кб	66,41 кб	3,91 кб

Таблиця 3.2 показує, що різниця в об'ємах пам'яті не занадто велика. При значному збільшенні кількості даних об'єм оперативної пам'яті все одно вимірюється кілобайтами одного порядку. Існують різні реалізації бінарного дерева, тому можна зменшити витрати пам'яті, але складність операцій також виросте. Якщо порахувати, то можна дізнатись, що для різниці в 1 мегабайт необхідно більше 260 тисяч елементів.

Таблиця 3.3 Кількість операцій порівняння для різних структур даних при проведенні основних операцій на них

	Операція порівняння					
	10 елементів			1000 елементів		
	Вставка	Пошук	Видалення	Вставка	Пошук	Видалення
Список	2	8	13	2	48	158
Дерево	5	3	7	19	31	21

Таблиця 3.4 Кількість операцій присвоєння для різних структур даних при проведенні основних операцій на них

	Операція присвоєння					
	10 елементів			1000 елементів		
	Вставка	Пошук	Видалення	Вставка	Пошук	Видалення
Список	7	8	16	7	48	161
Дерево	15	2	5	43	16	12

Таблиці показують , що для структури даних «Зв'язний список» менше операцій зроблено тільки при вставці. При операціях пошуку та видаленні на структурі «Бінарне дерево пошуку» очевидно менше і порівнянь і присвоєнь як для 10, так і для 1000 елементів.

Занесемо всі результати тестових досліджень в загальні таблиці, в яких зеленим кольором виділені кращі результати.

Таблиця 3. 5 Зведена таблиця порівняльного аналізу для телефонного довідника при 10 клієнтах

	10 елементів									
	Час виконання			Кількість операцій						Об'єм пам'яті
				Присвоєння			Порівняння			
	Вставка	Пошук	Видалення	Вставка	Пошук	Видалення	Вставка	Пошук	Видалення	
Список	18,68 нс	4,15 нс	8,3 нс	7	8	16	2	8	13	640 б
Дерево	33,21 нс	18,68 нс	21,79 нс	15	2	5	5	3	7	680 б

Таблиця 3. 6 Зведена таблиця порівняльного аналізу для телефонного довідника при 1000 клієнтах

	1000 елементів									
	Час виконання			Кількість операцій						Об'єм пам'яті
				Присвоєння			Порівняння			
	Вставка	Пошук	Видалення	Вставка	Пошук	Видалення	Вставка	Пошук	Видалення	
Список	0,61 мк	3,6 мк	6,74 мк	7	48	161	2	48	158	62,5 кб
Дерево	0,19 мк	0,84 мк	1,25 мк	43	16	12	19	31	21	66,41 кб

Асимптотичні оцінки алгоритмів

Таблиця 3.6 Асимптотичні оцінки основних алгоритмів на структурі даних «Зв'язний список»

В середньому			В найгіршому			
Пошук	Вставка	Видалення	Пошук	Вставка	Видалення	Витрати пам'яті
O(n)	O(1)	O(1)	O(n)	O(1)	O(1)	O(n)

Асимптотична оцінка O(1) означає, що трудомісткість операції не залежить від вхідних даних і є константою.

Для зв'язного списку швидко працюють вставка та видалення із початку або кінця списку. Пошук, вставка в середину та видалення з середини списку

мають вншу асимптотичну складність. Витрати пам’яті помірні. Додаткових затрат пам’яті немає. Вставка та видалення мається на увазі у початок списку. Адже складність вставки в обрану позицію списку рівносильна.

Таблиця 3.7 Асимптотичні оцінки основних алгоритмів на структурі даних «Бінарне дерево пошуку»

В середньому			В найгіршому			
Пошук	Вставка	Видалення	Пошук	Вставка	Видалення	Витрати пам'яті
O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)

Асимптотична оцінка O(n) означає, що трудомісткість операції залежить від вхідних даних і має лінійну від них залежність.

Асимптотична оцінка O(log n) означає, що трудомісткість операції залежить від вхідних даних і має логарифмічну від них залежність. В нотації теорвї алгоритмів основа логарифма є 2.

Для бінарного дерева пошуку в середньому швидко працюють операції пошуку, вставки та видалення. Затрати пам’яті також помірні. Але при виродженні дерева в лінійний список, тобто у найгіршому випадку всі операції будуть доволі довгими. Додаткових затрат пам’яті немає.

ВИСНОВКИ

В випускній роботі проведені дослідження по застосуванні структур даних «Зв'язний список» та «Бінарне дерево пошуку» в інформаційн0-комунікаційних застосуваннях прикладного характеру.

Комп'ютерне тестування обраних структур даних проводилось на задачі створення телефонного довідника. Було обрано дві реалізації – двонаправлений зв'язний список та бінарне дерево пошуку. Порівняльний аналіз проводився з метою визначення часу виконання основних стандартних операцій на цих структурах даних та кількості займаної пам'яті. Порівнювалось між собою також кількість операцій порівняння та присвоєння для обох структур.

Основні висновки по роботі.

1. Під час порівняння часу виконання було з'ясовано, що чим більший розмір вхідних даних, тим структура даних «Бінарне дерево пошуку» все більше виграє у всіх операціях над структурою «Зв'язний список».

Єдиний випадок коли це не так – при малій кількості даних – в цьому випадку «Зв'язний список» «виграє». Як висновок, при малій кількості даних в телефонному довіднику – 10, 100 людей, можна використовувати зв'язний список. А при великому об'ємі інформації доцільніше використовувати бінарне дерево пошуку.

2. При порівнянні було помічено невелику різницю в об'ємі використовуваної пам'яті. Структура даних «Бінарне дерево пошуку» використовує на 4 байти більше в порівнянні зі «Зв'язним списком». Різниці можна позбутися, якщо по-іншому реалізувати основу структури і основні алгоритми.

3. Досліджено – складність структури і її розуміння. Принцип побудови списку легкий, у ньому немає великих, складних для розуміння функцій. Дерево – навпаки, має багато розгалужень, через що і операції над вузлами дерева важко реалізовуються.

СПИСОК ЛІТЕРАТУРИ

1. Joshi B. K. Data Structures and Algorithms in C++ / Tata McGraw-Hill Education. – 2010 – 363 p.
2. Мелешко Є.В., Якименко М.С., Поліщук Л.І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. – Кропивницький: Видавець – Лисенко В.Ф., 2019. – 156 с.
3. Clifford A. Shaffer Data Structures and Algorithm Analysis, 2011. – p. 613. [Електронний ресурс] – Режим доступу до ресурсу: http://people.cs.vt.edu/˜shaffer/Book/errata.html.
4. Ален Б. Доуни Алгоритмы и структуры данных. Извлечение информации на языке Java. – СПб.:: Питер Пресс, 2018. – 240 с.
5. Онищенко В.В., Коник Р.С. Алгоритми і структури даних. – Київ, 2017. – 67 с.
6. Шаповалов С. П. Алгоритми і структури даних. – [Електронний ресурс] – Режим доступу до ресурсу: https://dl.sumdu.edu.ua/textbooks/95351/index.html
7. А. Бхаргава Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. - СПб.: Питер, 2017. - 288 с.

ДОДАТОК

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

struct nodeL {
	string name;
	string phone;
	nodeL* next;
	nodeL* prev;
};

class List {
private:
	nodeL* head;
	nodeL* tail;
	int size;
public:
	List() {
		head = tail = nullptr;
		size = 0;
	}
	~List() {
		deleteList();
	}
	int getSize() {
		return size;
```

```
}
nodeL* getPhone(string name) {
        nodeL* tmp = head;
        while (tmp->name != name)
                tmp = tmp->next;
        return tmp;
}
void addHead(string name, string phone) {
        nodeL* tmp = new nodeL;
        tmp->prev = nullptr;
        tmp->name = name;
        tmp->phone = phone;
        tmp->next = head;

        if (head != nullptr) head->prev = tmp;
        if (size == 0) head = tail = tmp;
        else head = tmp;
        size++;
}
void addTail(string name, string phone) {
        nodeL* tmp = new nodeL;
        tmp->next = nullptr;
        tmp->name = name;
        tmp->phone = phone;
        tmp->prev = tail;

        if (tail != nullptr) tail->next = tmp;
        if (size == 0) tail = head = tmp;
```

```
        else tail = tmp;
        size++;
}
void insert(int pos = 0) {
        if (!pos) {
                cout << "Enter position to insert: ";
                cin >> pos;
        }
        if (pos < 1 || pos > size + 1) {
                cout << "Incorrect position" << endl;
        }
        else {
                string name, phone;
                cout << "Enter name to insert: ";
                cin >> name;
                cout << "Enter phone number: ";
                cin >> phone;
                if (pos == 1) addHead(name, phone);
                else {
                        int i = 1;
                        nodeL* tmp = head;
                        while (i < pos) {
                                tmp = tmp->next;
                                i++;
                        }
                        nodeL* prev = tmp->prev;
                        nodeL* elem = new nodeL;
                        if (prev != nullptr && size != 1) prev->next = elem;
```

```
                elem->next = tmp;
                elem->prev = prev;
                tmp->prev = elem;
                size++;
            }
        }
    }
    void deleteName(string name) {
        nodeL* del = head;
        int i = 1;
        while (del->name != name) {
            del = del->next;
            i++;
        }

        nodeL* prev = del->prev;
        nodeL* next = del->next;

        if (prev != nullptr && size != 1) prev->next = next;
        if (next != nullptr && size != 1) next->prev = prev;
        if (i == 1) head = del->next;
        if (i == size) tail = del->prev;
        delete del;
        size--;
    }

    void deleteNode(int pos = 0) {
        if (!pos) {
```

```
            cout << "Enter position to insert: ";
            cin >> pos;
        }
        if (pos < 1 || pos > size + 1) {
            cout << "Incorrect position" << endl;
        }
        else {
            int i = 1;
            nodeL* del = head;
            while (i < pos) {
                del = del->next;
                i++;
            }

            nodeL* prev = del->prev;
            nodeL* next = del->next;
            if (prev != nullptr && size != 1) prev->next = next;
            if (next != nullptr && size != 1) next->prev = prev;
            if (pos == 1) head = next;
            if (pos == size) tail = prev;
            delete del;
            size--;
        }
    }
    void deleteList() {
        while (size) deleteNode(1);
    }
    void print() {
```

```
            if (size != 0) {
                    nodeL* tmp = head;
                    while (tmp != nullptr) {
                            cout << "Name: " << tmp->name << endl;
                            cout << "Phone: " << tmp->phone << endl << endl;
                            tmp = tmp->next;
                    }
            }
    }

struct nodeT {
    string name;
    string phone;
    nodeT* left;
    nodeT* right;
    nodeT* parent;
};

class Tree {
public:
    nodeT* root;
    Tree() {
            root = nullptr;
    }
    void insert(string name, string phone) {
            nodeT* tmp = new nodeT;
            tmp->name = name;
            tmp->phone = phone;
```

```
        tmp->left = nullptr;
        tmp->right = nullptr;

        nodeT* p;
        nodeT* t = nullptr;

        tmp->parent = tmp->left = tmp->right = nullptr;
        p = root;
        while (p != nullptr) {
                t = p;
                if (name < p->name) p = p->left;
                else p = p->right;
        }
        tmp->parent = t;
        if (t == nullptr) root = tmp;
        else {
                if (name < t->name) t->left = tmp;
                else t->right = tmp;
        }
}
nodeT* getNode(string name, nodeT* node) {
        if (node == nullptr || node->name == name) return node;

        if (node->name > name) return getNode(name, node->left);
        else return getNode(name, node->right);
}
void deleteNode(string name) {
        nodeT* del = getNode(name, root);
```

```
if (del->left == nullptr && del->right == nullptr) {
	if (del == root) {
		delete del;
		return;
	}
	if (del == del->parent->left) del->parent->left = nullptr;
	else del->parent->right = nullptr;
	delete del;
}
else if (del->left == nullptr && del->right != nullptr) {
	nodeT* tmp = del->right;
	if (del == root) root = del->right;
	else {
		if (del == del->parent->left) {
			tmp->parent = del->parent->left;
			del->parent->left = tmp;
		}
		else {
			tmp->parent = del->parent->right;
			del->parent->right = tmp;
		}
	}
	delete del;
}
else if (del->left != nullptr && del->right == nullptr) {
	nodeT* tmp = del->left;
	if (del == root) root = del->left;
	else {
```

```
                if (del == del->parent->left) {
                    tmp->parent = del->parent->left;
                    del->parent->left = tmp;
                }
                else {
                    tmp->parent = del->parent->right;
                    del->parent->right = tmp;
                }
            }
            delete del;
        }
        else {
            nodeT* tmp = del->right;
            if (tmp->left != nullptr) {
                while (tmp->left != nullptr)
                    tmp = tmp->left;
                if (tmp->right == nullptr) tmp->parent->left = nullptr;
                else {
                    tmp->right->parent = tmp->parent;
                    tmp->parent->left = tmp->right;
                }
            }
            else {
                if (tmp->right != nullptr) tmp->right->parent = tmp->parent;
                del->right = tmp->right;
            }

            del->name = tmp->name;
```

```
                del->phone = tmp->phone;
                delete tmp;
            }
        }
        void print(nodeT* node) {
            if (node != nullptr) {
                print(node->left);
                cout << "Name: " << node->name << endl;
                cout << "Phone: " << node->phone << endl << endl;
                print(node->right);
            }
        }
};

void menu(List l, Tree t) {
        cout << "Choose operation" << endl;
        cout << "Enter \"Add\" to add new element, \"Find\" to get info.\nEnter \"Inf
o\" to show all information, \"Delete\" to delete element, \"Exit\" to end program"
<< endl;
        string op;
        cin >> op;
        if (op == "Add") {
            cout << endl;

        cout << "Enter \"List\" to add to list. Enter \"Tree\" to add to tree. Enter \"Bot
h\" to add to both structures" << endl;
            cin >> op;
            if (op == "List") {
                string name, phone;
```

```
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter phone: ";
        cin >> phone;
        l.addHead(name, phone);
        cout << endl;
    }
    else if (op == "Tree") {
        string name, phone;
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter phone: ";
        cin >> phone;
        t.insert(name, phone);
    }
    else if (op == "Both") {
        string name, phone;
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter phone: ";
        cin >> phone;
        l.addHead(name, phone);
        t.insert(name, phone);
    }
    else {
        cout << "Incorrect operation" << endl;
        menu(l, t);
    }
```

```
	}
	else if (op == "Find") {
		cout << endl;

	cout << "Enter \"List\" to find in list. Enter \"Tree\" to find in tree. Enter \"Bot
h\" to find in both structures" << endl;
		cin >> op;
		if (op == "List") {
			string name;
			cout << "Enter name: ";
			cin >> name;
			nodeL *n = l.getPhone(name);
			cout << "Name: " << n->name << endl;
			cout << "Phone: " << n->phone << endl;
		}
		else if (op == "Tree") {
			string name;
			cout << "Enter name: ";
			cin >> name;
			nodeT* n = t.getNode(name, t.root);
			cout << "Name: " << n->name << endl;
			cout << "Phone: " << n->phone << endl;
		}
		else if (op == "Both") {
			string name;
			cout << "Enter name: ";
			cin >> name;
			nodeL* nl = l.getPhone(name);
```

```
                nodeT* nt = t.getNode(name, t.root);
                cout << "List" << endl;
                cout << "Name: " << nl->name << endl;
                cout << "Phone: " << nl->phone << endl << endl;
                cout << "Tree" << endl;
                cout << "Name: " << nt->name << endl;
                cout << "Phone: " << nt->phone << endl;
            }
            else {
                cout << "Incorrect operation" << endl;
                menu(l, t);
            }
        }
        else if (op == "Info") {
            cout << endl;

        cout << "Enter \"List\" to show list. Enter \"Tree\" to show tree. Enter \"Both\
" to show both structures" << endl;
            cin >> op;
            if (op == "List") {
                cout << "Phone book in list" << endl;
                l.print();
            }
            else if (op == "Tree") {
                cout << "Phone book in tree" << endl;
                t.print(t.root);
            }
            else if (op == "Both") {
```

```
                cout << "Phone book in list" << endl;
                l.print();
                cout << endl;
                cout << "Phone book in tree" << endl;
                t.print(t.root);
            }
            else {
                cout << "Incorrect operation" << endl;
                menu(l, t);
            }
        }
        else if (op == "Delete") {
            cout << endl;

        cout << "Enter \"List\" to delete from list. Enter \"Tree\" to delete from tree. E
nter \"Both\" to delete from both structures" << endl;
            cin >> op;
            if (op == "List") {
                string name, phone;
                cout << "Enter name: ";
                cin >> name;
                l.deleteName(name);
            }
            else if (op == "Tree") {
                string name, phone;
                cout << "Enter name: ";
                cin >> name;
                t.deleteNode(name);
```

```
            }
            else if (op == "Both") {
                string name, phone;
                cout << "Enter name: ";
                cin >> name;
                l.deleteName(name);
                t.deleteNode(name);
            }
            else {
                cout << "Incorrect operation" << endl;
                menu(l, t);
            }
        }
        else if (op == "Exit") {
            cout << "Work finished" << endl;
            exit(0);
        }
        else {
            cout << "Incorrect operation" << endl << endl;
            menu(l, t);
        }
        menu(l, t);
}

int main() {
        cout << "Enter name of the file with extension or full path" << endl;
        string f_name;
        cin >> f_name;
```

```
	ifstream in;
	in.open(f_name);
	if (!in.is_open()) {
		cout << "Error. File cannot be opened" << endl;
		exit(1);
	}
	cout << "Data from file " << f_name << " was added to list and tree" << endl << 
endl;

	string name, phone;
	List phoneList;
	Tree phoneTree;
	while (!in.eof()) {
		in >> name;
		in >> phone;
		phoneList.addHead(name, phone);
		phoneTree.insert(name, phone);
	}

	in.close();
	menu(phoneList, phoneTree);
	return 0;
}
```