**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ДВНЗ «УНІВЕРСИТЕТ БАНКІВСЬКОЇ СПРАВИ»**

**ІНСТИТУТ БАНКІВСЬКИХ ТЕХНОЛОГІЙ ТА БІЗНЕСУ**

**КАФЕДРА КІБЕРБЕЗПЕКИ ТА СОЦІАЛЬНИХ НАУК**

Спецiальність: «Кібербезпека»

Курс  **||** Група 203 Семестр **|**

**ЗАВДАННЯ**

**на курсову роботу студентки**

Тівоненко Аліни Андріївни

(прiзвище, iм`я, по батьковi)

1. Тема курсової роботи:

**Розробка та побудова алгоритів вирешення практичних завдань**

2. Термiн здачi студентом закiнченої роботи

3. Вихiднi данi до (роботи): інформаційно-аналітичні інтернет джерела, зразки

рішень практичних завдань та задач, нормативно-правова база, щодо оформлен-

ня технічної документації програмних продуктів, початкові значення для об-

робки.

4. Змiст пояснювальної записки (перелiк питань, які належить розробити)

Вступ. Формулювання вимог до програми. Проектування та практична

реалізація розв’язку поставленої задачі. Тестування програми і результати

її виконання. Висновки.

5. Перелiк графiчного матерiалу (з точним зазначенням обов`язкових креслень)

Таблиці та схеми представлено в роботі. Презентація додається.

6. Дата видачі завдання “ 10 ” жовтня 2019 р

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Назва етапiв курсової роботи | Термiн виконання етапiв курсової роботи | Примiт-ки |
| 1 | Вступ. |  |  |
| 2 | Розділ 1. Теоретичні аспекти теорії алгоритмів та структур даних  1.1. Питання 1  1.2. Питання 2  1.3. Питання 3  ……………………  1.N. Питання N |  |  |
| 3 | Розділ 2. Практична реалізація розв’язку поставлених задач  2.1. Оцінка коректності алгоритму та аналіз алгоритму на часову та просторову ефективності.  2.3. Опис програми (обґрунтування вибору мови програмування, опис основних процедур та функцій).  2.4. Опис змінних та їх ідентифікаторів, що використовуються в програмі.  2.6. Опис алгоритмів та кодів |  |  |
| 4 | Розділ 3. Тестування програми і результати її виконання  3.1. Опис контрольних прикладів.  3.2. Розробка настанови користувача |  |  |
| 5 | Висновки |  |  |
| 6 | Список використаних джерел. |  |  |
| 7 | Додатки. |  |  |
| 8 | Захист курсової роботи |  |  |

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (пiдпис)

Керiвник \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (пiдпис)

**ЗМІСТ**

ЗМІСТ…………………………………………………………………………….…..1

ВСТУП……………………………………………………………………………......2

РОЗДІЛ 1. Генеологічне дерево з використанням бінарного дерева…...........…..3

1.1. Дерево структура данних…..………………………………………………..3

1.2. Методи обходу дерева………….…………………………………………...4

1.3 Дерево в теорії графів………………………………………………………..5

1.4 Двійкове дерево пошуку………………………………………………….….7

1.5 Бінарне дерево пошуку………………………………………………….…..13

1.6 Рекурсія…………………………………………………………………….....14

1.7 Динамічні структури даних…………………………………………………16

РОЗДІЛ 2. Розробка програми виконання основного завдання………………....19

2.1. Розробка основного алгоритму виконання основного завдання………….19

2.2. Опис програми. Структура даних та її функції………………………..…19

РОЗДІЛ 3. Тестування програми та результати її виконання……………………21

3.1. Блок схема……………………………………………………………………21

3.2. Текст програми……………………………………………………………....22

3.3 Приклад програми……………………………………………………………24

ВИСНОВКИ……………………………………………………………………...…25

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ…………………………………………..25

**ВСТУП**

«C++» - це мова програмування загального призначення, високого рівня, яка була створена Б'ярном Страуструпом, як розширення мови програмування С. Мова значно розширилася з часом, а сучасний «С++» має об’єктно – орієнтовані, функціональні та загальні особливості. У 90 – х роках «С++» стала найуживанішою мовою загального призначення. Її використовували для розробки програмного забезпечення, написання драйверів, системного програмування й для розробок програм та ігор. Ця мова програмування стандартизована комітетом ISO за міжнародним стандартом, і час від часу покращується за допомогою додаванням нових приміток, пунктів та вимог.

Основними перевагами мови «С++» є:

* швидша дія роботи різних програм, ніж на мові програмування «С»;
* гнучкість. Можливість розробляти різні проекти для різноманітних платформ і систем;
* підтримка різних стилів та технологій програмування: ООП, традиційне директивне програмування, метапрограмування (шаблони, макроси), узагальнене програмування;
* має набір засобів (деструктори і конструктори, посилання, стандартні шаблони), що дозволяють майже повністю виключити виділення і звільнення пам'яті вручну і різні операції з вказівниками;
* існує підтримка як об'єктно-орієнтованого, так і процедурного програмування;
* має достатньо констант, шаблонів і вбудованих функцій.

1. **ГЕНЕОЛОГІЧНЕ ДЕРЕВО З ВИКОРИСТАННЯМ БІНАРНОГО ДЕРЕВА**
   1. **Дерево структура данних**

Дерево (можливо, нелінійне) — структура даних, яка складається з вузлів(вершин) і ребер, без будь-яких циклів. Дерево без вузлів називається нульовим або порожнім деревом. Дерево, яке не є порожнім, складається з кореневого вузла і багатьох рівнів додаткових вузлів, які утворюють [ієрархію](https://uk.wikipedia.org/wiki/%D0%86%D1%94%D1%80%D0%B0%D1%80%D1%85%D1%96%D1%8F), в [інформатиці](https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) та [програмуванні](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) одна з найпоширеніших [структур даних](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85). Формально [дерево визначається](https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2)) як скінченна множина Т з однією або більше вершин (вузлів, nodes), яка задовольняє наступним вимогам:

1. існує один відокремлений вузол — корінь (root) дерева
2. інші вузли (за винятком кореня) розподілені серед m ≥ 0 непересічних множин T1…Tm і кожна з цих множин, в свою чергу, є деревом. Дерева T1…Tm мають назву піддерев (subtrees) даного кореня.

Це найефективніший спосіб представлення і зберігання такої інформації.

З визначення випливає, що кожна [вершина](https://uk.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D1%88%D0%B8%D0%BD%D0%B0_(%D1%82%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2)) є в свою чергу коренем деякого піддерева. Кількість піддерев вершини має назву ступеня (degree) цієї вершини. Вершина ступеню нуль має назву кінцевої (terminal) або листа (leaf). Некінцева вершина також має назву вершини розгалуження (branch node).

Нехай x — довільна вершина дерева з коренем r. Тоді існує єдиний шлях з r до x. Усі вершини на цьому шляху називаються предками (ancestors) x; якщо деяка вершина y є предком x, то x називається нащадком (descendant) y. Нащадки та предки вершини x, що не збігаються з нею самою, називаються власними нащадками та предками. Кожну вершину x, в свою чергу, можна розглядати як корінь деякого піддерева, елементами якого є вершини-нащадки x.

Якщо вершина x є предком y та не існує вершин поміж ними (тобто x та y з'єднані одним ребром), а також існують предки для x (тобто x не є коренем), то вершина x називається батьком (parent) до y, а y — дитиною (child) x. Коренева вершина єдина не має батьків. Вершини, що мають спільного батька, називаються братами (siblings). Вершини, що мають дітей, називаються внутрішніми (internal). Глибиною вершини x називається довжина шляху від кореня до цієї вершини. Максимальна глибина вершин дерева називається висотою.

Піддерево — частина деревоподібної [структури даних](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85), яка може бути представлена ​​у вигляді окремого дерева. Будь-який вузол дерева T разом з усіма його вузлами-нащадками є піддеревом дерева T. Для будь-якого вузла піддерева або має бути шлях в кореневий вузол цього піддерева, або сам вузол повинен бути кореневим. Тобто піддерево пов'язано з кореневим вузлом цілого дерева, а відносини піддерева з усіма іншими вузлами визначаються через поняття відповідне піддерево (за аналогією з терміном «відповідна підмножина»).

Існує два основних типи дерев. У [рекурсивному](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D1%96%D1%8F) або невпорядкованому дереві має значення лише структура самого дерева без урахування порядку нащадків для кожного вузла. Дерево, в якому є заданий порядок (наприклад, кожному ребру, провідному до нащадка, присвоєні різні натуральні числа) називається деревом з іменованими ребрами або впорядкованим деревом зі структурою даних, заданої перед ім'ям. Впорядковані дерева є найбільш поширеними серед деревовидних структур. [Двійкове дерево пошуку](https://uk.wikipedia.org/wiki/%D0%94%D0%B2%D1%96%D0%B9%D0%BA%D0%BE%D0%B2%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83) — одне з різновидів упорядкованого дерева. Двійкове дерево — структура даних у вигляді дерева, в якому кожна вершина має не більше двох дітей. Зазвичай такі діти називаються правим та лівим. На базі двійкових дерев будуються такі структури, як [двійкові дерева пошуку](https://uk.wikipedia.org/wiki/%D0%94%D0%B2%D1%96%D0%B9%D0%BA%D0%BE%D0%B2%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83) та [двійкові купи](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D0%BD%D0%B0%D1%80%D0%BD%D0%B0_%D0%BA%D1%83%D0%BF%D0%B0). Існує безліч різних способів представлення дерев. Найбільш загальний спосіб представлення зображує вузли як записи, розташовані в [динамічно виділеній пам'яті](https://uk.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D1%96%D1%87%D0%BD%D0%B5_%D0%B2%D0%B8%D0%B4%D1%96%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F_%D0%BF%D0%B0%D0%BC%27%D1%8F%D1%82%D1%96) з вказівниками на своїх нащадків, предків (або і тих і інших), або, як елементи [масиву](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D0%B8%D0%B2), пов'язані між собою відносинами, визначеними їх позиціями в масиві (наприклад, двійкова купа).

* 1. **Методи обходу дерева**

Покроковий перебір елементів дерева по зв'язкам між вузлами-предками і вузлами-нащадками називається [обходом дерева](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%D1%85%D1%96%D0%B4_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%B0). Найчастіше, операція може бути виконана переходами вказівника на окремі вузли. Обхід, при якому кожен вузол-предок проглядається раніше його нащадків називається предвпорядкованим обходом або обходом в прямому порядку (pre-order walk), а коли проглядаються спочатку нащадки, а потім предки, то обхід називається післявпорядкованим обходом або обходом у зворотному порядку (post-order walk). Існує також симетричний обхід, при якому відвідується спочатку ліве піддерево, потім вузол, потім — праве піддерево, і обхід в ширину, при якому вузли відвідуються рівень за рівнем (N-й рівень дерева — безліч вузлів з висотою N). Кожен рівень обходиться зліва направо.

Обхід бінарного дерева передбачає відвідування усіх вершин [бінарного дерева](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D0%BD%D0%B0%D1%80%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), при цьому кожна з вершин відвідується тільки один раз. Існують три види таких обходів, кожний з яких визначається [рекурсивно](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D1%96%D1%8F):

* прямий порядок ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) preorder) наступної послідовності:

1. відвідати корінь
2. відвідати ліве піддерево
3. відвідати праве піддерево

Тобто, в такому порядку обходу кожна вершина відвідується до того, як будуть відвідані її діти.

* зворотний порядок ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *postorder*) наступної послідовності:

1. відвідати ліве піддерево
2. відвідати праве піддерево
3. відвідати корінь

Тобто, в такому порядку кожна вершина відвідується лише після того, як будуть відвідані її діти.

* центрований (центральний) порядок ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *inorder*) наступної послідовності:

1. відвідати ліве піддерево
2. відвідати корінь
3. відвідати праве піддерево

В такому порядку кожна вершина відвідується між відвіданням лівої та правої дитини. Такий порядок особливо часто застосовується в [бінарних деревах пошуку](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D0%BD%D0%B0%D1%80%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83), тому що дає можливість обходу вершин у порядку збільшення їхніх порядкових номерів.

|  |
| --- |
| [Бінарне дерево](https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Binary_tree_(oriented_digraph).png) |
| Мал.1 Бінарне дерево |

Для цього бінарного дерева:

* Прямий порядок: 2, 7, 2, 6, 5, 11, 5, 9, 4
* Зворотний порядок: 2, 5, 11, 6, 7, 4, 9, 5, 2
* Центрований (центральний) порядок: 2, 7, 5, 6, 11, 2, 5, 4, 9
  1. **Дерево в теорії графів**

Де́рево в [теорії графів](https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2) - зв'язний [граф](https://uk.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) без [циклів](https://uk.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_(%D1%82%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2)). Орієнтоване (спрямоване) дерево — ациклічний орграф (орієнтований граф, що не містить циклів) - той, в якому тільки одна вершина має нульову напівстепінь входу, а всі інші вершини мають напівстепінь входу 1. Вершина з нульовим степенем входу називається коренем дерева, вершини з нульовим напівстепенем виходу (з яких не виходить жодне ребро) називаються кінцевими вершинами або листям.

Формально дерево визначається як скінченна множина T одного або більше вузлів з наступними властивостями:

* 1. Існує один корінь дерева T
  2. Інші вузли (за вийнятком кореня) розподілені серед М  ≥ 0 непересічних множин T1…Tm  і кожна з множин є деревом, дерева T1…Tm  називається піддеревами даного кореня Т.

Найважливіші характеристичні властивості «дерева» висловлюються такими шістьма рівносильними одне одному висловленнями:

* {\displaystyle \varkappa (L)=1\,}x(L) = 1 та λ (L) = 0 (визначення «дерева»);
* λ (L) = 0 та m(L) = n(L) – 1;
* x(L) = 1 та m(L) = n(L) – 1
* для довільної пари вершин *x*, *y* в *L* існує один і тільки один [ланцюг](https://uk.wikipedia.org/wiki/%D0%9B%D0%B0%D0%BD%D1%86%D1%8E%D0%B3_(%D1%82%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2)), який з'єднує *x* та *y*;
* x(L) = 1, але але якщо із *L* видалити будь яке ребро, то для отриманого графу *L*− буде x(L− ) = 2;
* λ(L) = 0, але якщо до  {\displaystyle \lambda (L^{+})=1\,}L додати будь яке ребро (не додаючи вершин), то у отриманого графу L+ буде {\displaystyle \varkappa (L)=1\,}{\displaystyle \lambda (L)=0\,} λ(L+ ) = 1.

Тут L – довільний граф, n(L) – кількість його вершин , m(L) – кількість ребер, x(L) – кількість компонент зв'язності, λ(L) – цикломатичне число.

Довільний граф без циклів часто називають лісом (оскільки кожна його складова — «дерево»). Ордерево, яке росте із *x*0, — це «дерево», в якому виділено одну вершину *x*0 («корінь»), а ребра орієнтовані таким чином, що всі ланцюги, які починаються в *x*0, є шляхами (тобто, їхні дуги орієнтовані в напряму обходу).

[Степінь вершини](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BF%D1%96%D0%BD%D1%8C_%D0%B2%D0%B5%D1%80%D1%88%D0%B8%D0%BD%D0%B8_(%D1%82%D0%B5%D0%BE%D1%80%D1%96%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D1%96%D0%B2)) — кількість інцидентних їй ребер.

Кінцевий вузол (лист, термінальна вершина) — сайт зі ступенем 1 (тобто вузол, у який веде тільки одне ребро; у разі орієнтованого дерева — вузол, який веде тільки одна дуга і не виходить ні однієї дуги).

Вузол розгалуження — некінцевий вузол.

Рівень вузла  — довжина шляху від кореня до вузла. Можна визначити рекурсивно:

рівень кореня дерева дорівнює 0;

рівень будь-якого іншого вузла на одиницю більше, ніж рівень кореня найближчого піддерева дерева, що містить цей сайт.

Дерево із зазначеною вершиною називається кореневим деревом.

N-й ярус дерева — множина вузлів дерева, на n-ому рівні від кореня дерева.

Частковий порядок на вершинах: якщо вершини різні і вершина лежить на елементарному ланцюзі, що з'єднує корінь з вершиною кореневе дерево з коренем — підграф .

[Кістякове дерево](https://uk.wikipedia.org/wiki/%D0%9A%D1%96%D1%81%D1%82%D1%8F%D0%BA%D0%BE%D0%B2%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) (остов) — це підграф даного графа, що містить всі його вершини і є деревом. Ребра графа, що не входять в остов, називаються хордами графа відносно остова. Незведеним називається дерево, в якому немає вершин ступеня 2. Ліс — множина дерев, або незв'язний граф без циклів.

Властивості дерева графів:

* Дерево не має кратних ребер та петель.
* Будь-яке дерево з n {\displaystyle n} вершинами містить {\displaystyle n-1} *n - 1* ребер. Більш того, скінченний зв'язний граф є деревом, тоді і тільки тоді, коли *B - P {\displaystyle B-P=1}= 1*, де {\displaystyle B} *B* — число вершин, *P* {\displaystyle P}— число ребер графа.
* Граф є деревом, тоді і тільки тоді, коли будь-які дві різні його вершини можна з'єднати єдиним простим ланцюгом.
* Будь-яке дерево однозначно визначається відстаннями (найменшою довжиною ланцюга) між його кінцевими (ступеня 1) вершинами.
* Будь-яке дерево є двочастковим графом. Будь-яке дерево, множина вершин якого не більше ніж рахункова, є планарним графом.
* Для будь-яких трьох вершин дерева шляхи між парами цих вершин мають одну спільну вершину.
  1. **Двійкове дерево пошуку**

В [програмуванні](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) двійкове дерево — [структура даних](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85) у вигляді [дерева](https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)), в якому кожна вершина має не більше двох дітей. Зазвичай такі діти називаються правим та лівим. На базі двійкових дерев будуються такі структури, як [двійкові дерева пошуку](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D0%BD%D0%B0%D1%80%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83) та [двійкові купи](https://uk.wikipedia.org/wiki/%D0%91%D1%96%D0%BD%D0%B0%D1%80%D0%BD%D0%B0_%D0%BA%D1%83%D0%BF%D0%B0).

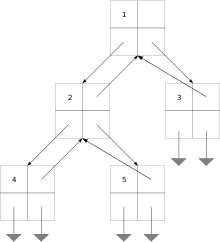
* Двійкове дерево — таке кореневе дерево, в якому кожна вершина має не більше двох дітей.
* Повне (закінчене) двійкове дерево — таке двійкове дерево, в якому кожна вершина має нуль або двох дітей.
* Ідеальне двійкове дерево — це таке повне двійкове дерево, в якому листя (вершини без дітей) лежать на однаковій глибині (відстані від кореня).

Двійкове дерево на кожному *n*-му рівні має від 1 до 2n вершин.

Часто виникає необхідність обійти усі вершини дерева для аналізу інформації, що в них знаходиться. Існують декілька порядків такого обходу, кожний з яких має певні властивості, важливі в тих чи інших алгоритмах: прямий (preorder), центрований (inorder) та зворотний (postorder).

Залежно від задач, які вирішуються цими структурами та можливостей тої чи іншої [мови програмування](https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B2%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F), існує декілька варіантів конструювання двійкових дерев.

Реалізація з використанням [вказівників](https://uk.wikipedia.org/wiki/%D0%92%D0%BA%D0%B0%D0%B7%D1%96%D0%B2%D0%BD%D0%B8%D0%BA) передбачає зберігання в кожній вершині дерева x, разом із даними власне цієї вершини, також двох полів - правого та лівого (right[x] та left[x]), які містять вказівники на відповідних дітей цієї вершини.



Мал 2. Змінена реалізація двійкового дерева. Кожна вершина містить також вказівник на батьківську вершину.

Також іноді додається вказівник p[x] на батьківську вершину. Це спрощує деякі алгоритми та виявляється корисним, коли необхідно забезпечити швидкий доступ до батьківської вершини. Іноді достатньо тільки вказівника на батьківську вершину. Взагалі будь-яке орієнтоване дерево можна описати, знаючи тільки зв'язки від дітей до батьківської вершини.

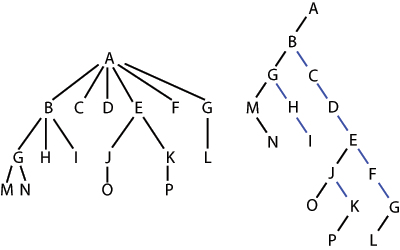
Деякі різновиди двійкових дерев (наприклад, [червоно-чорні дерева](https://uk.wikipedia.org/wiki/%D0%A7%D0%B5%D1%80%D0%B2%D0%BE%D0%BD%D0%BE-%D1%87%D0%BE%D1%80%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) або [AVL-дерева](https://uk.wikipedia.org/wiki/AVL-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE)), вимагають збереження в вершинах і деякої додаткової інформації. Якщо у вершини відсутня одна чи обидві дитини, відповідні вказівники ініціалізуються спеціальними «пустими» значеннями.

https://upload.wikimedia.org/wikipedia/commons/thumb/8/86/Binary_tree_in_array.svg/220px-Binary_tree_in_array.svg.png

Мал.3 Двійкове дерево на базі масиву

Двійкові дерева також можуть бути побудовані на базі [масивів](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D0%B8%D0%B2_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)). Такий метод набагато ефективніший щодо економії пам'яті. В такому представленні, якщо вершина має порядковий номер i, то її діти знаходяться за індексами 2i+1 та 2i+2, а батьківська вершина за індексом ((i-1)/2) (за умов, що коренева вершина має індекс 0).Інший варіант зберігання дерева в масиві — зберігати індекси дітей.

Існує єдине та взаємооднозначне відображення довільного впорядкованого дерева в двійкове.

[](https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:Nary_to_binary_tree_conversion.png)

Мал. 4 Представлення n-арних дерев як двійкових

Для цього слід послідовно зв'язати усіх дітей кожної сім'ї з першою дитиною та видалити усі вертикальні з'єднання за виключенням з'єднання батька з першою дитиною в сім'ї. Тобто кожна вершина N впорядкованого n-арного дерева відповідає вершині M деякого двійкового дерева. Ліва дитина вершини M відповідає першій дитині вершини N, а права дитина M відповідає першому з наступних братів N (тобто першому з наступних дітей батька вершини N).

Така відповідність має назву природної відповідності між n-арним та двійковим деревом.

Двійкове (або Бінарне) дéрево пóшуку ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) binary search tree, BST) в [інформатиці](https://uk.wikipedia.org/wiki/%D0%86%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) — [двійкове дерево](https://uk.wikipedia.org/wiki/%D0%94%D0%B2%D1%96%D0%B9%D0%BA%D0%BE%D0%B2%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), в якому кожній вершині x зіставлене певне значення val[x]. При цьому такі значення повинні задовольняти умові впорядкованості:

* нехай x — довільна вершина двійкового дерева пошуку. Якщо вершина y знаходиться в лівому піддереві вершини x, то val[y] ≤ val[x].
* Якщо у знаходиться у правому піддереві x, то val[y] ≥ val[x].

Таке структурування дозволяє надрукувати усі значення у зростаючому порядку за допомогою простого алгоритму [центрованого обходу дерева](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%D1%85%D1%96%D0%B4_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%B0).[[джерело?](https://uk.wikipedia.org/wiki/%D0%92%D1%96%D0%BA%D1%96%D0%BF%D0%B5%D0%B4%D1%96%D1%8F:%D0%9F%D0%BE%D1%81%D0%B8%D0%BB%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BD%D0%B0_%D0%B4%D0%B6%D0%B5%D1%80%D0%B5%D0%BB%D0%B0)]

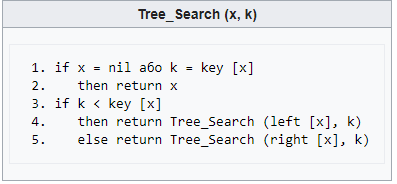
Представляється таке дерево вузлами наступного вигляду:

\*Node = (element, key, left, right, parent). Доступ до дерева T здійснюється за допомогою посилання root.

Бінарні дерева пошуку набагато ефективніші в операціях пошуку, аніж лінійні структури, в яких витрати часу на пошук пропорційні [O(n)](https://uk.wikipedia.org/wiki/%D0%9D%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F_%D0%9B%D0%B0%D0%BD%D0%B4%D0%B0%D1%83), де n — розмір [масиву](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%81%D0%B8%D0%B2_(%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)) даних, тоді як в повному бінарному дереві цей час пропорційний в середньому O(log2n) або O(h), де h — висота дерева (хоча гарантувати, що h не перевищує log2n можна лише для [збалансованих дерев](https://uk.wikipedia.org/wiki/%D0%97%D0%B1%D0%B0%D0%BB%D0%B0%D0%BD%D1%81%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), які є ефективнішими в алгоритмах пошуку, аніж прості бінарні дерева пошуку).

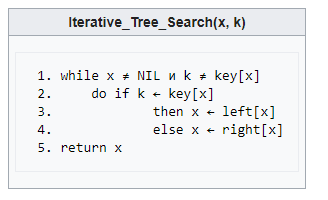
Найпоширенішою операцією, яка виконується з бінарним деревом пошуку, є пошук в ньому певного ключа. Крім того, бінарні дерева пошуку підтримують такі запити, як пошук мінімального і максимального елемента, а також попереднього і наступного.

Для пошуку вузла із заданим ключем в бінарному дереві пошуку використовується наступна [процедура](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) 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).

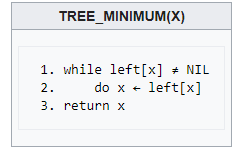


[Процедура](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) пошуку починається з кореня дерева і проходить вниз по дереву. Для кожного вузла *х* на шляху вниз його ключ *key[x]* порівнюється з переданим як параметр ключем *k*. Якщо ключі однакові, пошук завершується. Якщо *k* менше *key[х]*, пошук триває в лівому піддереві *х*; якщо більше — то пошук переходить в праве піддерево. Ту ж процедуру можна записати [ітеративно](https://uk.wikipedia.org/wiki/%D0%86%D1%82%D0%B5%D1%80%D0%B0%D1%86%D1%96%D1%8F), «розгортаючи» [рекурсію](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D1%96%D1%8F) в цикл while.

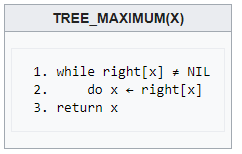
### Ітеративна версія процедури Пошук



### Пошук мінімального (максимального) елемента. Алгоритм пошуку мінімального елемента. Елемент з мінімальним значенням ключа легко знайти, слідуючи за вказівниками *left* від кореневого вузла до тих пір, поки не зустрінеться значення NIL. Процедура TREE\_MINIMUM(x) повертає покажчик на знайдений елемент піддерева з коренем *x*.

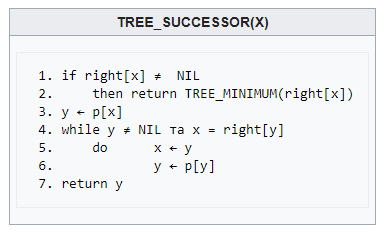


Алгоритм пошуку максимального елемента симетричний:



Обидва алгоритми вимагають часу [*O(h)*](https://uk.wikipedia.org/wiki/%D0%9D%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F_%D0%9B%D0%B0%D0%BD%D0%B4%D0%B0%D1%83), де *h* — висота дерева.

Наступний і попередній елементи. Якщо *x* — покажчик на деякий вузол дерева, то процедура TREE\_SUCCESSOR(X) повертає покажчик на вузол з наступним за *x* елементом або nil, якщо зазначений елемент — останній в дереві:

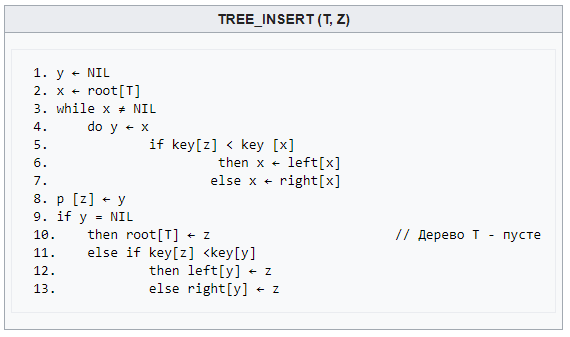


Наведена процедура окремо розглядає два випадки. Якщо праве піддерево вершини не пусте, то наступний за *x* елемент є крайнім лівим вузлом у правому піддереві, який виявляється процедурою TREE\_MlNlMUM(right[x]). З іншого боку, якщо праве піддерево вузла *x* пусте, та у *x* існує наступний за ним елемент *y*, то *y* є найменшим предком *x*, чий лівий вузол також є предком *x*. Для того щоб знайти *y*, ми просто піднімаємося вгору по дереву до тих пір, поки не зустрінемо вузол, який є лівим дочірнім вузлом свого батька. Ця дія виконується в рядках 3-7 алгоритму.

Час роботи алгоритму TREE\_SUCCESSOR в дереві заввишки *h* складає [*O(h)*](https://uk.wikipedia.org/wiki/%D0%9D%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F_%D0%9B%D0%B0%D0%BD%D0%B4%D0%B0%D1%83), оскільки ми або рухаємося по шляху вниз від вихідного вузла, або по шляху нагору. Процедура пошуку подальшого вузла в дереві TREE\_PREDECESSOR симетрична процедурі TREE\_SUCCESSOR і також має час роботи [*O(h)*](https://uk.wikipedia.org/wiki/%D0%9D%D0%BE%D1%82%D0%B0%D1%86%D1%96%D1%8F_%D0%9B%D0%B0%D0%BD%D0%B4%D0%B0%D1%83).

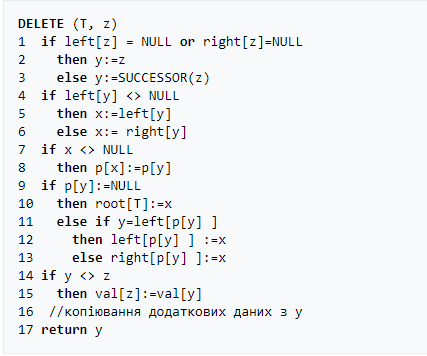
Якщо в дереві є вузли з однаковими ключами, ми можемо просто визначити наступний і попередній вузли як ті, що повертаються процедурами TREE\_SUCCESSOR та TREE\_PREDECESSOR відповідно.

Для вставки нового значення *v* в бінарне дерево пошуку *Т* 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 *Т* і деякі поля *z*, що *z* виявляється вставленим в відповідну позицію в дереві.



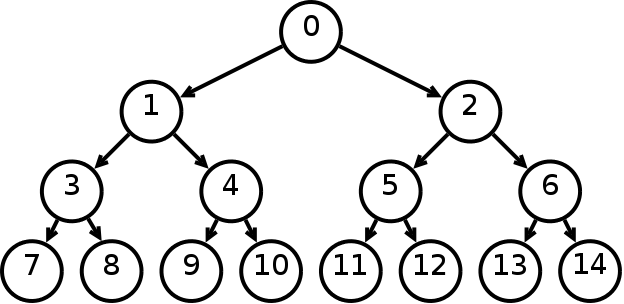
Процедура видалення даного вузла *z* з бінарного дерева пошуку отримує як аргумент покажчик на *z*. Процедура розглядає три можливі ситуації:

1. Якщо у вузла *z* немає дочірніх вузлів, то ми просто змінюємо його батьківський вузол *р[z]*, замінюючи в ньому покажчик на *z* значенням NIL.
2. Якщо у вузла *z* лише один дочірній вузол, то ми видаляємо вузол *z*, створюючи новий зв'язок між батьківським і дочірнім вузлом вузла *z*.
3. Якщо у вузла *z* два дочірніх вузла, то ми знаходимо наступний за ним вузол *у*, у якого немає лівого дочірнього вузла, прибираємо його з позиції, де він перебував раніше, шляхом створення нового зв'язку між його батьком і нащадком, і замінюємо ним вузол *Z*.



* 1. **Бінарне дерево пошуку**

Бінарне дерево - це ієрархічна структура даних, в якій кожен вузол має значення (воно ж є в даному випадку і ключем) і посилання на лівого і правого нащадка. Вузол, що знаходиться на самому верхньому рівні (який не є чиїмось нащадком) називається коренем. Вузли, які не мають нащадків (обидва нащадка яких дорівнюють NULL) називаються листям.

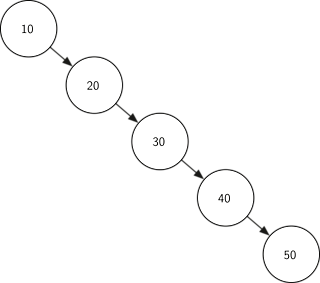


мал.4 Бінарне дерево

Бінарне дерево пошуку - це бінарне дерево, що володіє додатковими властивостями: значення лівого нащадка менше значення батька, а значення правого нащадка більше значення батька для кожного вузла дерева. Тобто, дані в бінарному дереві пошуку зберігаються в відсортованому вигляді. При кожній операції вставки нового або видалення існуючого вузла відсортованого порядок дерева зберігається. При пошуку елемента порівнюється шукане значення з коренем. Якщо шукане більше кореня, то пошук продовжується в правому нащадку кореня, якщо менше, то в лівому, якщо одно, то значення знайдено і пошук припиняється.

Збалансоване бінарне дерево пошуку - це бінарне дерево пошуку з логарифмічною висотою. Дане визначення скоріше ідейне, ніж суворе. Суворе визначення оперує різницею глибини найглибшого і самого неглибокого листа (в AVL-деревах) або відношенням глибини найглибшого і самого неглибокого листа (в червоно-чорних деревах). У збалансованому бінарному дереві пошуку операції пошуку, вставки і видалення виконуються за логарифмічна час (так як шлях до будь-якого листу від кореня не більше логарифма). У виродженим випадку незбалансованого бінарного дерева пошуку, наприклад, коли в порожнє дерево вставлялася відсортована послідовність, дерево перетвориться в лінійний список, і операції пошуку, вставки і видалення будуть виконуватися за лінійний час. Тому балансування дерева вкрай важлива. Технічно балансування здійснюється поворотами частин дерева при вставці нового елементу, якщо вставка даного елемента порушила умова збалансованості.

Збалансоване бінарне дерево пошуку застосовується, коли необхідно здійснювати швидкий пошук елементів, що чергується зі вставками нових елементів і вилученнями існуючих. У разі, якщо набір елементів, що зберігається в структурі даних фіксований і немає нових вставок і вилучень, то масив краще. Тому що пошук можна здійснювати алгоритмом бінарного пошуку за той же логарифмічна час, але відсутні додаткові витрати по зберіганню і використанню покажчиків. Наприклад, в С ++ асоціативні контейнери set і map представляють собою збалансоване бінарне дерево пошуку.



мал. 5 Екстремально незбалансоване бінарне дерево пошуку

**1.6 Рекурсія**

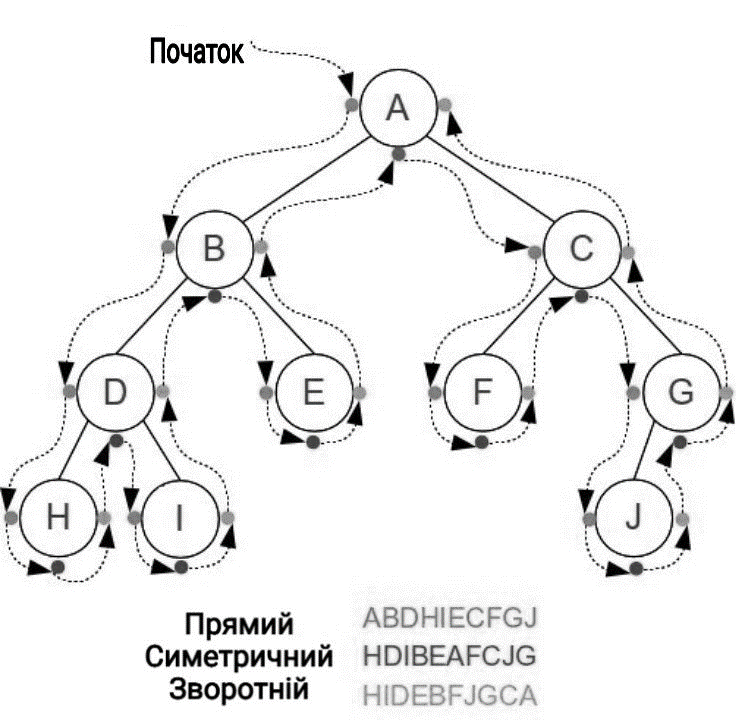
Рекурсія в програмуванні - це виклик функцією самої себе з іншими аргументами. В принципі, рекурсивна функція може викликати сама себе і з тими ж самими аргументами, але в цьому випадку буде нескінченний цикл рекурсії, який закінчиться переповненням стека. Усередині будь-рекурсивної функції повинен бути базовий випадок, при якому відбувається вихід з функції, а також виклик або виклики самої себе із іншими аргументами. Аргументи не просто повинні бути іншими, а повинні наближати виклик функції до базового нагоди. Наприклад, виклик всередині рекурсивної функції розрахунку факторіала повинен йти з меншим за значенням аргументом, а виклики всередині рекурсивної функції обходу дерева повинні йти з вузлами, що перебувають далі від коріння, ближче до листя. Рекурсія може бути не тільки прямий (виклик безпосередньо себе), але і непрямої. Наприклад А викликає Б, і Б викликає О. За допомогою рекурсії можна емулювати ітеративний цикл, і навіть роботу структури даних стек (LIFO).

Коротко обговоримо дерева з точки зору теорії графів. Граф - це безліч вершин і ребер. Ребро - це зв'язок між двома вершинами. Кількість можливих ребер в графі квадратично залежить від кількості вершин (для розуміння можна уявити турнірну таблицю зіграних матчів). Дерево - це зв'язний граф без циклів. Можливості підключення означає, що з будь-якої вершини в будь-яку іншу існує шлях по ребрах. Відсутність циклів означає, що даний шлях - єдиний. Обхід графа - це систематичне відвідування всіх його вершин по одному разу кожної. Існує два види обходу графа: 1) пошук в глибину; 2) пошук в ширину.

З формальної точки зору можна зробити як рекурсивну, так і ітеративну версію як пошуку в ширину, так і пошуку в глибину. Для обходу в ширину в обох випадках необхідна чергу. Рекурсія в рекурсивної реалізації обходу в ширину всього лише емулює цикл. Для обходу в глибину існує рекурсивна реалізація без стека, рекурсивна реалізація зі стеком і ітеративна реалізація зі стеком. Ітеративна реалізація обходу в глибину без стека неможлива.

Прямий обхід йде в наступному порядку: корінь, лівий нащадок, правий нащадок. Симетричний - лівий нащадок, корінь, правий нащадок. Зворотний - лівий нащадок, правий нащадок, корінь. У коді рекурсивної функції відповідного обходу зберігається відповідний порядок викликів (порядок рядків коду), де замість кореня йде виклик даної рекурсивної функції.

Якщо нам дано зображення дерева, і потрібно знайти його обходи, то може допомогти наступна техніка (мал. 3). Обводимо дерево обвідної замкнутої кривої (починаємо йти зліва вниз і замикаємо справа вгору). Прямому обходу буде відповідати порядок, в якому огинає, рухаючись від кореня вперше проходить поруч з вузлами зліва. Для симетричного обходу порядок, в якому огинає, рухаючись від кореня вперше проходить поруч з вузлами знизу. Для зворотного обходу порядок, в якому огинає, рухаючись від кореня вперше проходить поруч з вузлами справа. У коді рекурсивного виклику прямого обходу йде: виклик, лівий, правий. Симетричного - лівий, виклик, правий. Зворотного - лівий правий, виклик.



мал. 6 Допоміжний малюнок для обходів

Для бінарних дерев пошуку симетричний обхід проходить всі вузли в відсортованому порядку. Якщо ми хочемо відвідати вузли в назад відсортованому порядку, то в коді рекурсивної функції симетричного обходу слід поміняти місцями правого і лівого нащадка.

**1.7 Динамічні структури даних**

 Динамічні структури за визначенням характеризуються відсутністю фізичної суміжності елементів структури в пам'яті непостійністю і непередбачуваністю розміру (числа елементів) структури в процесі її обробки.

Оскільки елементи динамічної структури розташовуються по непередбачуваним адресами пам'яті, адреса елемента такої структури не може бути обчислений з адреси початкового або попереднього елемента. Для встановлення зв'язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв'язки між елементами.  Таке представлення даних в пам'яті називається зв'язковим. Елемент динамічної структури складається з двох полів:

• інформаційного поля або поля даних, в якому містяться ті дані, заради яких і створюється структура; в загальному випадку інформаційне поле саме є інтегрованою структурою - вектором, масивом, інший динамічною структурою і т. п. ;

• поле зв'язок, в якому містяться один або кілька покажчиків, що зв'язує даний елемент з іншими елементами структури;

Коли зв'язне подання даних використовується для розв'язання прикладної задачі, для кінцевого користувача "видимим" робиться тільки вміст інформаційного поля, а поле зв'язок використовується тільки програмістом-розробником.

Переваги зв'язкового представлення даних - в можливості забезпечення значною мінливості структур;

• розмір структури обмежується тільки доступним об'ємом машинної пам'яті;

• при зміні логічної послідовності елементів структури потрібно не переміщення даних в пам'яті, а тільки корекція покажчиків;

• велика гнучкість структури.

Разом з тим чіткий подання не позбавлене й недоліків, основні з яких:

• на поля зв'язок споживання пам'ять;

• доступ до елементів зв'язковий структури може бути менш ефективним за часом.

Останній недолік є найбільш серйозним і саме їм обмежується застосовність зв'язкового представлення даних.  Якщо в суміжному поданні даних для обчислення адреси будь-якого елементу нам у всіх випадках достатньо було номера елемента і інформації, що міститься в дескрипторі структури, то для зв'язкового подання адресу елемента не може бути обчислений з вихідних даних.  Дескриптор зв'язковий структури містить один або кілька покажчиків, що дозволяють увійти в структуру, далі пошук необхідного елемента виконується проходженням по ланцюжку покажчиків від елемента до елемента.  Тому зв'язне уявлення практично ніколи не застосовується в задачах, де логічна структура даних має вид вектора або масиву - з доступом за номером елемента, але часто застосовується в задачах, де логічна структура вимагає іншої вихідної інформації доступу (таблиці, списки, дерева і т. д).

Списком називається впорядкована множина, що складається із змінного числа елементів, до яких застосовні операції включення, виключення. Список, що відображає відносини сусідства між елементами, називається лінійним.  Довжина списку дорівнює числу елементів, що містяться в списку, список нульової довжини називається порожнім списком.  Лінійні зв'язні списки є найпростішими динамічними структурами даних.

Графічно зв'язку в списках зручно зображувати за допомогою стрілок.  Якщо компонента не пов'язана ні з якою іншою, то в поле покажчика записують значення, не вказує ні на який елемент.  Таке посилання позначається спеціальним ім'ям - nil.

возв’язний список характеризується наявністю пари покажчиків в кожному елементі: на попередній елемент і на наступний:

Очевидний плюс тут у тому, що від цього елемента структури ми можемо піти в обидві сторони.  Таким чином спрощуються багато операцій.  Однак на покажчики витрачається додаткова пам'ять.

Різновидом розглянутих видів лінійних списків є кільцевої список, який може бути організований на основі як однозв’язного, так і двозв'язного списків. При цьому в однозв’язного списку покажчик останнього елемента повинен вказувати на перший елемент; в двозв'язному списку в першому і останньому елементах відповідні покажчики перевизначаються.

При роботі з такими списками кілька спрощуються деякі процедури.  Однак, при перегляді такого списку слід прийняти деяких запобіжних заходів, щоб не потрапити в нескінченний цикл.

Описувані нижче АТД можуть бути організовані на базі

1.  масиву: виділяється місце під N елементів разом, а потім описуються операції над даним типом даних в термінах операцій над елементами масиву.

2.  списку: пам'ять виділяється та звільняється в міру необхідності.

Перший варіант швидше, але лише другий істинно динамічний.  Відповідно, в різних додатках може бути кращий перший (розмір структури відомий і невеликий) або другий (розмір заздалегідь невідомий).  Ми будемо розглядати переважно динамічні рішення.

Стек - такий послідовний список зі змінною довжиною, включення і виключення елементів з якого виконуються тільки з одного боку списку, званого вершиною стека.  Застосовуються й інші назви стека - магазин і черга, яка функціонує за принципом LIFO (Last - In - First-Out - "останнім прийшов - першим виключається")

 Приклади стека: гвинтівковий патронний магазин, тупиковий залізничний роз'їзд для сортування вагонів.

Основні операції над стеком - включення нового елемента (англійська назва push - заштовхувати) і виключення елемента зі стека (англ. pop - вискакувати).

Корисними можуть бути також допоміжні операції:

• визначення поточного числа елементів в стеку;

• очищення стека;

• неруйнуюче читання елемента з вершини стека, яке може бути реалізоване, як комбінація основних операцій:

 x: = pop (stack); push (stack, x);

Деякі автори розглядають також операції включення / виключення елементів для середини стека, проте структура, для якої можливі такі операції, не відповідає стеку за визначенням.

Для наочності розглянемо невеликий приклад, що демонструє принцип включення елементів в стек і виключення елементів з стека.  На рис.  4 (а, б, с) зображені стану стека:

• а).  порожнього;

• б-г).  після послідовного включення в нього елементів з іменами 'A', 'B', 'C';

• д, е).  після послідовного видалення з стека елементів 'C' і 'B';

• ж).  після включення в стек елементу 'D'.

Стек можна уявити, наприклад, у вигляді стопки книг (елементів), що лежить на столі.  Привласнимо кожній книзі свою назву, наприклад A, B, C, D. . .  Тоді в момент часу, коли на столі книг немає, про стек аналогічно можна сказати, що він порожній, тобто не містить жодного елемента.  Якщо ж ми почнемо послідовно класти книги одну на іншу, то отримаємо стопку книг (припустимо, з n книг), або отримаємо стек, в якому міститься n елементів, причому вершиною його буде елемент n+1. Видалення елементів з стека здійснюється аналогічним чином тобто видаляється послідовно по одному елементу, починаючи з вершини, або по одній книзі з стопки. **Черга FIFO**

Чергою FIFO (First - In - First-Out - "першим прийшов - першим виключається").  називається такий послідовний список зі змінною довжиною, в якому включення елементів виконується тільки з одного боку списку (цю сторону часто називають кінцем або хвостом черги), а виняток - з іншого боку (званої початком або головою черги).  Ті самі черги до прилавків і до кас, які ми так не любимо, є типовим побутовим прикладом черги FIFO.

Основні операції над чергою - ті ж, що і над стеком - включення, виключення, визначення розміру, очищення, неруйнуюче читання.

Динамічна реалізація черги аналогічна реалізації стека. Дек - особливий вид черги.  Дек (від англ. Deq - double ended queue, т. е чергу з двома кінцями) - це такий послідовний список, в якому як включення, так і виключення елементів може здійснюватися з будь-якого з двох кінців списку.  Окремий випадок дека - дек з обмеженим входом і дек з обмеженим виходом.  Логічна і фізична структури дека аналогічні логічної і фізичної структурі кільцевої FIFO-черзі.  Однак, стосовно до деку доцільно говорити не про початок і кінець, а про лівому і правому кінці.

Операції над Деком:

• включення елемента справа;

• включення елемента зліва;

• виключення елемента справа;

• виключення елемента зліва;

• визначення розміру;

• очищення.

Фізична структура дека в статичній пам'яті ідентична структурі кільцевої черги.  Динамічна реалізація є очевидним об'єднанням стека і черги.

Задачі, що вимагають структури дека, зустрічаються в обчислювальній техніці і програмуванні набагато рідше, ніж завдання, реалізовані на структурі стека або черги.  Як правило, вся організація дека виконується програмістом без яких-небудь спеціальних засобів системної підтримки

Прикладом дека може бути, наприклад, якийсь термінал, в який вводяться команди, кожна з яких виконується якийсь час.  Якщо ввести наступну команду, не дочекавшись, поки закінчиться виконання попередньої, то вона встане в чергу і почне виконуватися, як тільки звільниться термінал.  Це FIFO чергу.  Якщо ж додатково ввести операцію скасування останньої введеної команди, то виходить дек.

**2. РОЗРОБКА ПРОГРАМИ ВИКОНАННЯ ОСНОВНОГО ЗАВДАННЯ**

## **2.1 Розробка основного алгоритму виконання основного завдання**

Для виконання поставленого завдання, потрібно написати структуру геологічного дерева за допомогою динамічної структури , для пошуку всіх нащадків та їх вік.

Щоб створити геологічне дерево , потрібно скористатись структурою бінарного дерева. З самого напочатку на вершині йде кореневий обєкт дерева , який називається корнем і на нього указується \*root , а вершина яким закінчується називається «листком». Вся структура указана в блок схемі.

Для реалізації такої динамічної структури С++ , оприділим дві структури.

# Перша структура буде називатися PERSON і буде зберігати імя і вік.

# Друга структура OBJ буде представляти об’єкт бінарного дерева, вона буде включати в себе першу структуру PERSON (дані), і два покажчика : left , right.

## **2.2 Опис програми. Структура даних та її функції**

В програмі описана структура PERSON та OBJ. Задаємо перечислення typedef, воно буде зберігати дві константи: OBJ\_LEFT, OBJ\_RIGHT. За ним визначений глобальний покажчик на корінь дерева і йому присвоюється значення NULL - це буде означати що бінарне дерево пусте, в ньому немає ніяких обєктів. Потім, прописую функцію добавлення об’єкта бінарного дерева буде називатися add\_obj, і повертати покажчик на новий створений об’єкт OBJ\* . В самій функції спочатку створюю новий об’єкт , на нього буде вказуватися ptr, у нового створеного об’єкта. Далі потрібно зробити видалення обєкта з бінарного дерева, для цього використовую рекрусивну фунцію del\_obj , але запускатися вона буде в функції del\_obj\_all в якості аргументу цієї функції (OBJ\* obj) .

В самій функції роблю перевірку. Рекурсивна функція перевіряє, якщо покажчик obj = NULL то функція завершується, якщо не дорівнює нулю , то запускаємо рекурсивні функції. Также аналогічно працює функція від поразки рекурсивної функції , вона відображає всі об’єкти бінарного дерева. За ним йде функція main() , яка добавляє об’єкти в бінарне дерево в саму функцію show\_obj та перевіряємо, якщо об’єкт до якого ми можемо переходити далі існує , то він не дорівнює 0 , то виводимо інформацію по ньому на екрані та побачимо інформацію, потім запускаємо вершину по рекурсі , по лівій частині та правій. Спочатку підемо по лівій вершині відобразивши інформацію "Mom Grandma" , далі йти нікуди функція не піде завдяки умові ( if (obj == NULL) return) вона завершить роботу, і об’єкт повернеться до "Mother", потім йде по правій частині і повертається знову до "Mother". Після цього об’єкт переходе на ''Father '', і проводить аналогічну дію. Повернувшись до корневої вершини виконуємо функцію :

show\_obj(root->left);

show\_obj(root->right);

Потім відбувається видалення бінарного дерева del\_obj\_all(root) вона видаляє корневу вершину delete root.

* typedef struct{} – динамічна структура , typedef створює синонім для типу.
* typedef struct tag\_obj – придставляє об’єкт бінарного дерева, вона включає в себе першу структуру.
* OBJ\* add\_obj – це функція добавлення об’єкта бінарного дерева, називається add\_obj і повертає покажчик на новий створений OBJ\*.
* void del\_obj – рекрусивна фунція, яка робить видалення об’єкта з бінарного дерева , для цього будемо використовувати рvoid del\_obj\_all.
* void show\_obj – рекурсивна функція працює таким чином, вона теж перевіряє якщо покажчик obj = NULL то функція завершується.
* void show\_tree – аналогічно працює ф-ія від поразки рекурсивної ф-ії вона відображає всі обєкти бінарного дерева.
* Int main() - викликається при старті програми після ініціалізації нелокальних об'єктів зі статичної тривалістю зберігання. Функція яка добавляє об'єкти в це бінарне дерево.
* show\_obj – функція перевіряє об’єкт який переходить далі, якщо існує то дорівнює 0, і виводить її на екран.
* del\_obj\_all(root) – видалення бінарного дерева.
* delete root – видаляє кореневу вершину.
* return ptr – повертає покажчик на новий створений об'єкт.

**3**.**ТЕСТУВАННЯ ПРОГРАМИ І РЕЗУЛЬТАТИ ЇЇ ВИКОНАННЯ**

* 1. **Блок схема**

name, old

root\*

I

\*pm , Mother

\*pp

Dad

Grandmo

Grandfa

Grandmo

Grandfa

leftright

left right left right

left right left right left right left right

NULL NULL NULL NULL NULL

* 1. **Текст програми**

#include <iostream>

using namespace std;

typedef struct {

char name[100];

short old;

}PERSON;

typedef struct tag\_obj {

PERSON p;

struct tag\_obj\* left, \* right;

} OBJ;

typedef enum type\_obj\_tree { OBJ\_LEFT, OBJ\_RIGHT } OBJ\_TYPE;

OBJ\* root = NULL;

OBJ\* add\_obj(OBJ\* obj, PERSON& p, OBJ\_TYPE type = OBJ\_LEFT) {

OBJ\* ptr = new OBJ;

ptr->left = ptr->right = NULL;

ptr->p = p;

if (obj == NULL) root = ptr;

else {

switch (type) {

case OBJ\_RIGHT: obj->right = ptr; break;

case OBJ\_LEFT: obj->left = ptr;

}

}

return ptr;

}

void del\_obj(OBJ\* obj) {

if (obj == NULL) return;

del\_obj(obj->left);

del\_obj(obj->right);

cout << "Delete:" << obj->p.name << endl;

delete obj;

}

void del\_obj\_all(OBJ\* obj)

{

if (obj == NULL) return;

del\_obj(obj->left);

del\_obj(obj->right);

obj->right = obj->left = NULL;

}

void show\_obj(OBJ\* obj) {

if (obj == NULL) return;

cout << obj->p.name << "" << obj->p.old << endl;

show\_obj(obj->left);

show\_obj(obj->right);

}

void show\_tree() {

if (root == NULL) return;

cout << root->p.name << "" << root->p.old << endl;

show\_obj(root->left);

show\_obj(root->right);

}

int main() {

PERSON pr = { "Alina", 18 };

add\_obj(NULL, pr);

strcpy\_s(pr.name, "Mother");

pr.old = 38;

OBJ\* pm = add\_obj(root, pr);

strcpy\_s(pr.name, "Dad");

pr.old = 42;

OBJ\* pp = add\_obj(root, pr, OBJ\_RIGHT);

strcpy\_s(pr.name, "Mom Grandma");

pr.old = 62;

add\_obj(pm, pr);

strcpy\_s(pr.name, "Mom Grandfather");

pr.old = 62;

add\_obj(pm, pr, OBJ\_RIGHT);

strcpy\_s(pr.name, "Dad Grandma");

pr.old = 72;

add\_obj(pp, pr);

strcpy\_s(pr.name, "Dad Grandfather");

pr.old = 85;

add\_obj(pp, pr, OBJ\_RIGHT);

show\_tree();

cout << endl;

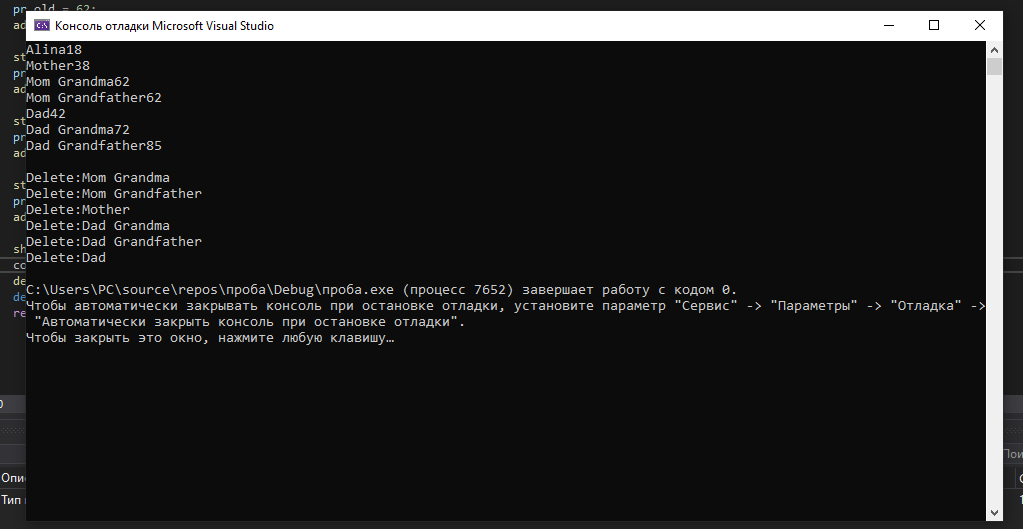
del\_obj\_all(root);

delete root;

return 0;

}

* 1. **Приклад програми**



**ВИСНОВОК**

Під час розробки програми мною були засвоєні основні функції мови C++. Розробила програму-генеологічне дерево використовуючи функції, розмішені всередині структури, що описує відповідний об’єкт. В програмі застосовані способи та прийоми роботи з вказівниками та різноманітними функціями. Основна задача полягає у виконанні елементарних дій та виведення. Дана програма дозволяє зручно виконувати дії . Також неможна не згадати, що завдяки написанню даної курсової роботи я отримала хороший практичний досвід у використанні функцій програмування.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

* + 1. Habr  ["procs"](https://finclub.net/ua/all/author/285-%22finansovyypulys%22.html) [Електронний ресурс] [Веб-сайт]. – Електронні дані.   [habr.com](mailto:info@finclub.net)  – Режим доступу: <https://habr.com/ru/post/267855/> (дата звернення 10.12.2019) – Назва з екрана.
    2. ALLREF  [Електронний ресурс] [Веб-сайт]. – Електронні дані.   [<https://allref.com.ua/uk/contacts.php>](mailto:info@finclub.net)  – Режим доступу: <https://allref.com.ua/uk/skachaty/Dinamichni_strukturi_danih_%28S++%29> (дата звернення 11.12.2019) – Назва з екрана.
    3. Т. А. Павловская «С/С++. Программирование на языке высокого уровня», учебник для ВУЗов, Питер, 2007. - 462с.
    4. Т. А. Павловская, Ю. А. Щупак «С/С++ Структурное программирование. Практикум» учебное пособие, Питер, 2007. - 238с.
    5. Н. Культин «С/С++ в задачах и примерах», Санкт-Петербург, «БХВ-Петербург», 2004. -281с.
    6. [Енциклопедія кібернетики](https://uk.wikipedia.org/wiki/%D0%95%D0%BD%D1%86%D0%B8%D0%BA%D0%BB%D0%BE%D0%BF%D0%B5%D0%B4%D1%96%D1%8F_%D0%BA%D1%96%D0%B1%D0%B5%D1%80%D0%BD%D0%B5%D1%82%D0%B8%D0%BA%D0%B8), [Зиков О. О.](https://uk.wikipedia.org/wiki/%D0%97%D0%B8%D0%BA%D0%BE%D0%B2_%D0%9E%D0%BB%D0%B5%D0%BA%D1%81%D0%B0%D0%BD%D0%B4%D1%80_%D0%9E%D0%BB%D0%B5%D0%BA%D1%81%D0%B0%D0%BD%D0%B4%D1%80%D0%BE%D0%B2%D0%B8%D1%87), т. 1, с. 256.
    7. [Дональд Кнут](https://uk.wikipedia.org/wiki/%D0%94%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%B4_%D0%9A%D0%BD%D1%83%D1%82), Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М.: «Вильямс», 2006. — С. 720.