Лабораторна робота № 2

Теоретичні відомості. Динамічні структури даних. Нелінійні списки (дерева): загальні положення

Деревом називається зв'язний граф без простих циклів (це одне з багатьох означень) [1].

Дерево можна також означити як рекурсивну структуру вузлів, яка є або порожньою, або складається з базового вузла – *кореня* (див. далі), з яким зв'язана скінчена кількість дерев, які ще називають *піддеревами* (див. далі) [4, 7].

Якщо дві вершини у дереві суміжні, то одну з них називають *батьком*, а іншу *сином*. За необхідності, ребро від батька до сина можна вважати орієнтованим. Вершина, що не має батька називається *коренем* дерева. Вершини, що не мають синів називаються *листками*. Вершини, які мають синів називаються *внутрішніми* (корінь також належить до внутрішніх вершин). Якщо існує шлях (орієнтований) з однієї вершини дерева в іншу, то одна з цих вершин є *предком*, а інша – *нащадком* (батько і син є частковим випадком). Нехай вершина не є листком. Тоді підграф, що містить цю вершину і всіх її нащадків називається *піддеревом* із коренем в цій вершині.

Кількість синів вершини v називають її *степенем* (позн. $\deg(v)$). Дерево називається *m-арним*, якщо жодна вершина має не більше ніж m синів ($\deg(v) \le m$). Якщо кожна вершина має точно m синів ($\deg(v) = m$), то дерево називається *повним m-арним*.

При $m = 2$ розглядають *бінарне* та *повне бінарне* дерева. У бінарному дереві для вершини, що має двох синів розрізняють *лівого сина* та *правого сина*. Якщо вершина має тільки одного сина, то його вважають лівим. У бінарному дереві для вершин, що не є листками аналогічно вводиться поняття *лівого піддерева* та *правого піддерева*.

Рівнем вершини *L* називається довжина шляху від кореня до цієї вершини (рівень кореня рівний нулеві). *Висотою* дерева *H* називають довжину найдовшого шляху між вершинами в ньому (тобто максимальний рівень). Дерево називається *збалансованим*, якщо всі його листки знаходяться на одному, або сусідніх рівнях (тобто рівнях *H* та *H* – 1).

В даній роботі будемо розглядати тільки бінарні дерева.

У мові *C/C++* вузол (вершина) бінарного дерева може бути оголошений через таку структуру:

```
struct Node
{
datatype key; // Інформаційне поле (ключ) вузла
Node* parent; // Вказівник на батьківський вузол
Node* left; // Вказівник на лівого сина
Node* right; // Вказівник на правого сина
};
```

Для листків значення вказівників `left` та `right` приймають значення `NULL`. Для кореня значення `NULL` приймає вказівник `parent`.

Найпоширенішими операціями для роботи з деревами є: створення дерева, додавання вузла у дерево, видалення вузла з дерева, обхід дерева та пошук у ньому.

Створення бінарного дерева.

Розглянемо рекурсивний алгоритм створення бінарного дерева з наперед заданою кількістю вузлів *n* [5]. Даний алгоритм дозволяє створити симетричне бінарне дерево з мінімальною, для заданої кількості вузлів, глибиною. На кожному кроці рекурсії кількість вузлів у лівому і правому піддереві визначається за формулами: $n_{left} = n/2$, $n_{right} = n - n_{left} - 1$.

Рекурсивний алгоритм створення бінарного дерева (функція `CreateTree(…)`*).*

1. Перевірити умову зупинки рекурсії: чи кількість вузлів рівна нулеві. В цьому випадку дерево порожнє і *вказівник на поточний вузол* `pNode` потрібно встановити `NULL` (`*pNode = NULL`).
2. Інакше:
   2.1. Створити новий вузол дерева: (`*pNode = new Node`). Заповнити інформаційне поле нового вузла даними: `((*pNode)->key = data)`. Встановити вказівники нового вузла на ліве і праве піддерево як `NULL`: `((*pNode)->left = (*pNode)->right = NULL)`.
   2.2. Рекурсивно створити ліве піддерево даного вузла з кількістю вузлів (`n_left = n/2`).
   2.3. Рекурсивно створити праве піддерево даного вузла з кількістю вузлів (`n_right = n - n_left - 1`).

Зауваження. Рекурсивні виклики функції `CreateTree(…)`, що реалізує даний алгоритм повинні містити як параметри вказівники на ліве/праве піддерево та кількість вузлів у лівому/правому піддереві. Початковими значеннями цих параметрів є *вказівник на корінь дерева* `root` (ініціалізується значенням `NULL`) та загальна кількість вузлів дерева.

Зауваження. В даному алгоритмі вказівник вузла `parent` не використовується.

Відображення бінарного дерева.

Розглянемо тепер рекурсивний алгоритм відображення щойно створеного бінарного дерева. Піддерево рівня *L* виводиться так: спочатку рекурсивно відображається ліве піддерево, потім корінь піддерева рівня *L*, перед яким потрібно вивести *L* пробілів, потім відображається праве піддерево.

Рекурсивний алгоритм відображення дерева (функція `ShowTree(…)`*).*

1. Починаємо з кореня дерева (`pNode = root`). Поки не досягнуто листків (`pNode != NULL`):
   1.1. Вивести ліве піддерево.
   1.2. Вивести (в циклі) *L* пробілів та ключ вузла *L*–го рівня (`pNode->key`).
   1.3. Вивести праве піддерево.

Зауваження. Функція `ShowTree(…)`, що реалізує даний алгоритм повинна містити як параметри вказівник на ліве/праве піддерево та номер рівня (якщо розглядається *L*–ий рівень, то функція рекурсивно викликається для рівня *L*+1). Початковими значеннями цих параметрів є вказівник на корінь дерева `root` та нульовий рівень (*L* = 0).

Обхід дерева.

Задача обходу дерева полягає у одноразовому відвідуванні всіх вузлів дерева та виконання з ними певних дій (в нашому випадку – вивід на екран значення інформаційного поля (ключа) вузла). З іншого боку цю задачу можна розглядати як задання порядку (впорядкування) на множині вершин дерева.

Існує три рекурсивних алгоритми обходу дерева:

- обхід у прямому порядку (зверху вниз);
- обхід у зворотному порядку (знизу вверх);
- обхід у внутрішньому порядку (зліва направо).

Рекурсивний алгоритм обходу дерева у прямому порядку (функція `PrefixOrder(…)`*).*

1. Починаємо з кореня дерева (`pNode = root`). Поки не досягнуто листків (`pNode != NULL`):
   1.1. Обробити дані у вузлі (вивести на екран);
   1.2. Здійснити (рекурсивно) обхід лівого піддерева;
   1.3. Здійснити (рекурсивно) обхід правого піддерева;

Рекурсивний алгоритм обходу дерева у зворотному порядку (функція `PostfixOrder(…)`*).*

1. Починаємо з кореня дерева (`pNode = root`). Поки не досягнуто листків (`pNode != NULL`):
   1.1. Здійснити (рекурсивно) обхід лівого піддерева;
   1.2. Здійснити (рекурсивно) обхід правого піддерева;
   1.3. Обробити дані у вузлі (вивести на екран);

Рекурсивний алгоритм обходу дерева у внутрішньому порядку (функція `InfixOrder(…)`*).*

1. Починаємо з кореня дерева (`pNode = root`). Поки не досягнуто листків (`pNode != NULL`):
   1.1. Здійснити (рекурсивно) обхід лівого піддерева;
   1.2. Обробити дані у вузлі (вивести на екран);
   1.3. Здійснити (рекурсивно) обхід правого піддерева;

Зауваження. Одним із застосувань обходу дерев є побудова різних форм запису виразів (див. лабораторну роботу „Форми запису арифметичних виразів” з курсу „Дискретна математика”).

Хід роботи:

Частина 1. Побудова та обхід бінарного дерева

1. Створити нову бібліотеку `Tree` (файли `Tree.h, Tree.cpp`).
2. У файлі `Tree.h` оголосити структуру `Node` що задає вузол дерева, та за допомогою команди `typedef` зв’язати тип інформаційного поля вузла дерева `datatype` з типом даних, заданим викладачем.
3. У бібліотеці `Tree`, згідно описаних вище алгоритмів, реалізувати функції `CreateTree(…)` та `ShowTree(…)` для створення та відображення бінарного дерева. У цьому ж файлі реалізувати функції

обходу дерева: `PrefixOrder(…)`, `PostfixOrder(…)` та `InfixOrder(…)`

4. Створити новий проект `Lab_1_1` та підключити до нього бібліотеку `Tree`. У функції `main()` проекту реалізувати меню для виконання операцій створення, відображення та обходу дерева.
5. Відкомпілювати проект та продемонструвати його роботу для даних нелінійного списку (дерева), отриманих від викладача.

Бінарні дерева пошуку

Бінарним деревом пошуку (англ. *binary search tree* (BST)) називається бінарне дерево, що задовольняє таким умовам:

- поля даних (ключі) у різних вузлах не можуть бути однакові;
- для будь-якого вузла поля даних (ключі) вузлів його лівого піддерева є меншими, а правого – більшими за ключ цього вузла.

Подібна організація даних є зручною з точки зору їх пошуку. Алгоритмічна складність процедури пошуку у бінарному дереві пошуку становить $O(\log(n))$, де n – кількість вузлів дерева.

Розглянемо алгоритми для створення бінарного дерева пошуку, пошуку даних у ньому, додавання та видалення вузлів.

Пошук даних за ключем у бінарному дереві пошуку.

Спочатку реалізуємо процедуру пошуку за ключем, оскільки вона використовується як при додаванні вузлів у дерево, так і при їх видалені.

Алгоритм пошуку даних у бінарному дереві пошуку за ключем пошуку (функція `SearchNodeBST(…)`*).*

1. Починаємо з кореня дерева (`pNode = root`). Поки не досягнуто листків (`pNode != NULL`) або не знайдені дані:
   1.1. Якщо ключ поточного вузла співпадає з ключем пошуку, то дані знайдено;

1.2. Інакше, якщо ключ поточного вузла більший за ключ пошуку, то перейти до лівого піддерева (`pNode = pNode->left`);
1.3. Інакше, якщо ключ поточного вузла менший за ключ пошуку то перейти до правого піддерева (`pNode = pNode->right`);

Зауваження. Функція `SearchNodeBST(…)`, що реалізує даний алгоритм повинна містити як параметр ключ, за яким здійснюється пошук даних та повертати вказівник на знайдений вузол з даними (`NULL`, якщо вузол не знайдено, тобто досягнуто листка).

Створення (додавання нових вузлів) бінарного дерева пошуку.

Процедура створення бінарного дерева пошуку складається з трьох етапів: створення першого вузла (кореня), пошуку місця вставки і безпосередньо самої вставки нового вузла.

Алгоритм створення першого вузла (кореня) у бінарному дереві пошуку (функція `CreateRootBST(…)`*).*

1. Виділити динамічну пам’ять для нового вузла: (`Node *pNode = new Node`).
2. Заповнити інформаційне поле нового вузла: `(pNode->key)` даними.
3. Встановити вказівники нового вузла на батьківський вузол, ліве і праве піддерево як `NULL`: `(pNode->parent = pNode->left = pNode->right = NULL)`.

Зауваження. Функція, що реалізує даний алгоритм повинна повертати вказівник на новостворений елемент – корінь дерева `root`.

Алгоритм вставки нового вузла у бінарне дерево пошуку (функція `InsertNodeBST(…)`*).*

1. Виконати пошук за ключем даних, які потрібно вставити (використати функцію `SearchNodeBST(…)`). Якщо вузол з такими даними існує – видати відповідне повідомлення і припинити виконання функції.
2. Інакше: виділити динамічну пам'ять для нового вузла: (`Node *pNew = new Node`). Заповнити інформаційне поле нового вузла `(pNew->key)` даними. Встановити вказівники нового вузла на батьківський вузол, ліве і праве піддерево як `NULL`: `(pNew->parent = pNew->left = pNew->right = NULL)`.
3. *Пошук місця для вставки даних*. Починаємо з кореня дерева (`pNode = root`). Поки не досягнуто листків (`pNode != NULL`):
   3.1. Ввести додатковий вказівник `previous` на попередній вузол. Встановити значення цього вказівника рівним `pNode`;
   3.2. Інакше, якщо ключ поточного вузла більший за ключ пошуку, то перейти до лівого піддерева (`pNode = pNode->left`);
   3.3. Інакше, якщо ключ поточного вузла менший за ключ пошуку то перейти до правого піддерева (`pNode = pNode->right`);
4. *Вставка даних*:
   4.1. Встановити вузол `previous` батьківським для нового вузла (`pNew->parent = previuos`);
   4.2. Якщо дані батька поточного вузла `pNode` більші за ключ (`previous->key > key`) то приєднати новий вузол як його (батька) лівого сина (`previous->left = pNew`);
   4.3. Якщо дані батька поточного вузла менші за ключ (`previous->key < key`) то приєднати новий вузол як його (батька) правого сина (`previous->right = pNew`);

Видалення вузла бінарного дерева пошуку.

При видаленні вузла у бінарному дереві пошуку можливі три випадки.

1. Вузол, що видаляється, не має нащадків (тобто є листком). В цьому випадку покажчику батька на вузол, що видаляється, потрібно присвоїти значення `NULL` та звільнити динамічну пам’ять від даного вузла.

2. Вузол, що видаляється має одного нащадка. В цьому випадку покажчику батька на вузол, що видаляється, потрібно присвоїти адресу нащадка вузла, що видаляється. Після цього можна видалити потрібний вузол.

3. Вузол, що видаляється має двох нащадків. Це є найскладніший випадок. У цьому разі, на місце вузла, що видаляється, потрібно переставити вузол, що містить ключ, який є наступний (англ. *successor*) або попередній (англ. *predecessor*) у відсортованій послідовності ключів. Тобто в дереві не існує вузла з ключем, який є більшим за ключ вузла, що видаляється та менший за ключ вузла Successor. Аналогічно, в дереві не існує вузла з ключем, який є меншим за ключ вузла, що видаляється та більший за ключ вузла Predecessor. Вузли Successor та Predecessor ще називають *термінальними*. Алгоритми пошуку вузлів Successor та Predecessor описано нижче. У реальній реалізації видалення вузла з двома нащадками його ключу присвоюється значення ключа вузла Successor (або Predecessor), після чого відповідний термінальний вузол видаляється. Тобто, насправді видаляється не вузол з двома нащадками (у нього тільки змінюється ключ), а вузол Successor (або Predecessor). Термінальний вузол завжди є листком або вузлом з одним нащадком і процедура їх видалення описана вище.

Алгоритм пошуку термінального вузла Successor (функція `SuccessorNodeBST(…)`).

Зауваження. Дана функція повинна містити як параметр адресу вузла (`pNode`), для якого шукається вузол Successor та повертати адресу цього термінального вузла (`NULL`, якщо не знайдено).

1. Якщо у вузла `pNode` існує правий син (праве піддерево):
   1.1. Перейти у корінь правого піддерева вузла `pNode`: встановити додатковий вказівник `previous` на вузол правого сина (`previous = pNode->right`).
   1.2. Поки існує лівий син (`previous->left != NULL`): перейти до лівого піддерева (`previous = previous->left`)
   1.3. Повернути адресу вузла `previous` як значення вузла Successor для вузла `pNode`.

Зауваження. Кроки 1.1–1.3 є алгоритмом пошуку вузла з мінімальним ключем у піддереві з коренем `pNode->right` і можуть бути реалізовані як окрема функція (напр. `MinimumNodeBST(…)`) для пошуку вузла з мінімальним ключем для будь якого піддерева з коренем, який передається як параметр цієї функції (у випадку всього дерева цей параметр повинен приймати значення `root`).

Зауваження. Алгоритм пошуку вузла з максимальним ключем є аналогічним попередньому з незначними модифікаціями і може бути реалізований як окрема функція (напр. `MaximumNodeBST(…)`).

2. Інакше: якщо у вузла `pNode` не існує правого сина (правого піддерева):
   2.1. Перейти у батьківський вузол вузла `pNode` (`previous = pNode->parent`);
   2.2. Поки існує батьківський вузол (`previous != NULL`) та поточний вузол (`pNode`) є правим сином для цього батьківського вузла (`pNode == previous->right`): піднімаємось деревом вгору ліворуч (`pNode = previous; previous = previous->parent`);
   2.3. Повернути адресу вузла `previous` як значення вузла Successor для початкового значення вузла `pNode`.

Зауваження. Кроки 2.1–2.3 здійснюють переміщення по дереву вгору ліворуч від початкового значення вузла `pNode` допоки це можливо. Після

цього здійснюється поворот вгору праворуч і перший вузол буде вузлом Successor для початкового значення вузла `pNode`. Якщо вузол `pNode` є лівим сином батьківського вузла `previous`, то крок 2.2 пропускається і значення `previous` є значенням вузла Successor для вузла `pNode`. Якщо вузол `pNode` є коренем (`root`), то значення вузла Successor = `NULL`.

Алгоритм пошуку термінального вузла Predecessor (функція `PredecessorNodeBST(…)`).

Алгоритм пошуку термінального вузла Predecessor є аналогічним алгоритму пошуку термінального вузла Successor з незначними модифікаціями, зокрема пошуком вузла з максимальним ключем у піддереві з коренем у вузлі `pNode->left` (кроки 1.1–1.3) та переміщенням по дереву вгору праворуч від початкового значення вузла `pNode` допоки це можливо та наступним поворотом вгору ліворуч (кроки 2.1–2.3).

Алгоритм видалення вузла з бінарного дерева пошуку (функція `DeleteNodeBST(…)`).

Зауваження. Дана функція повинна містити як параметр адресу вузла (`*delNode`), що видаляється та, за необхідності, може повертати ключ цього вузла для подальшої обробки. Перед викликом функції `DeleteNodeBST(…)` потрібно здійснити пошук вузла `delNode` (функція `SearchNodeBST(…)`) та видати повідомлення, якщо такого вузла не знайдено.

Випадок 1. Вузол, що видаляється не має нащадків (листок):

1. Якщо вузол, що видаляється є коренем (`*delNode = *root`): видалити корінь (`delete *root`) та завершити виконання функції;
2. Інакше:
   2.1. Якщо вузол `delNode` – лівий син батьківського вузла (`(((*delNode)->parent)->left == *delNode`), то встановити

вказівник лівого сина цього батьківського вузла як `NULL` (`((*delNode)->parent)->left = NULL`);

2.2. Інакше: (якщо `delNode` – правий син батьківського вузла `(((*delNode)->parent)->right == *delNode)`) встановити вказівник правого сина як `NULL` (`((*delNode)->parent)->right = NULL`).

3. Звільнити пам'ять від вузла `delNode` (`delete *delNode`).

Випадок 2. Вузол, що видаляється має одного нащадка:

1. Визначити, якого нащадка має вузол, що видаляється (`delNode`) – лівого чи правого:
   1.1. Якщо вузол `delNode` має лівого нащадка (`(*delNode)->left != NULL`), то присвоїти вузлу-нащадку (`next`) вказівник лівого сина `delNode` (`next = (*delNode)->left`);
   1.2. Інакше: присвоїти вузлу-нащадку вказівник правого сина `delNode` (`next = (*delNode)->right`).
2. Якщо вузол, що видаляється є коренем (`*delNode == *root`): встановити вузол-нащадок як корінь дерева (`*root = next`), звільнити пам'ять від вузла `delNode` (`delete *delNode`).
3. Інакше:
   3.1. Якщо вузол `delNode` – лівий син батьківського вузла `(((*delNode)->parent)->left == *delNode)`, то встановити вказівник лівого сина цього батьківського вузла на вузол-нащадок (`((*delNode)->parent)->left = next`);
   3.2. Інакше: (якщо `delNode` – правий син батьківського вузла `(((*delNode)->parent)->right == *delNode)`) встановити вказівник правого сина батьківського вузла на вузол-нащадок (`((*delNode)->parent)->right = next`);
   3.3. Зв'язати вузол–нащадок з батьківським вузлом `delNode` (`next->parent = (*delNode)->parent`).

3.4.Звільнити пам’ять від вузла `delNode` (`delete *delNode`).

Випадок 3. Вузол, що видаляється має обох нащадків:

1. Знайти термінальний вузол для вузла `delNode` (`term = SuccessorNodeBST(*delNode)` або `term = PredecessorNodeBST (*delNode)`);
2. Оновити ключ вузла `delNode` значенням ключа термінального вузла (`(*delNode)->key = term->key`);
3. Видалити термінальний вузол, рекурсивно викликавши для нього функцію `DeleteNodeBST(…)`.

Хід роботи:

Частина 2. Бінарне дерево пошуку

1. У бібліотеці `Tree`, згідно описаних вище алгоритмів, реалізувати функції `SearchNodeBST(…)`, `CreateRootBST(…)`, `InsertNodeBST(…)`, `DeleteNodeBST(…)`, `SuccessorNodeBST(…)`, `PredecessorNodeBST(…)` для виконання операцій пошуку даних, створення кореневого вузла, додавання, видалення та пошуку термінальних вузлів у бінарному дереві пошуку.
2. Створити новий проект `Lab_1_2` та підключити до нього бібліотеку `Tree`. У функції `main()` проекту реалізувати меню для виконання операцій, описаних у п.1. Після кожної операції вставки/видалення вузла потрібно викликати функцію `ShowTree(…)`, для відображення змін у бінарному дереві пошуку.
3. Відкомпілювати проект та продемонструвати його роботу для даних бінарного дерева пошуку, отриманих від викладача.

Перелік посилань

1. Нікольский Ю. В. Дискретна математика / Ю. В. Нікольский, В. В. Пасічник, Ю. М. Щербина. – Львів: „Магнолія–2006”, 2013. – 432 с.
2. Андерсон Д. Дискретная математика и комбинаторика / Д Андерсон. – М.: Изд. дом „Вильямс”, 2004. – 960 с.
3. Кормен Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзер, Р. Ривест, К. Штайн. – М.: Изд. дом „Вильямс”, 2011. – 1296 с.
4. Вирт Н. Алгоритмы и структуры данных / Н. Вирт. – М.: Мир, 1989. – 360 с.
5. Ковалюк Т. В. Алгоритмізація та програмування / Т. В. Ковалюк. – Львів: „Магнолія–2006”, 2013. – 400 с.
6. Павловская Т. А. C/C++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб.: Изд-во „Питер”, 2013. – 461 с.
7. Ахо А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Изд. дом „Вильямс”, 2010. – 400 с.
8. Липский В. Комбинаторика для программистов / В. Липский. – М.: Мир, 1988. – 216 с.
9. Седжвик Р. Алгоритмы на C++ / Р. Седжвик. – М.: Изд. дом „Вильямс”, 2014. – 1056 с.