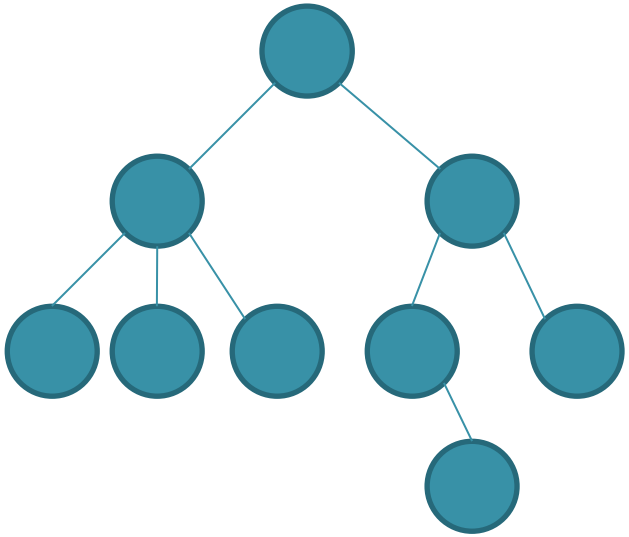
Лекція 10.

Дерева. Основні операції з деревами.

§1. Кореневі дерева. Основні поняття

Дерево – це сукупність елементів, що називаються вузлами (один з яких корінь), та відношень („батьківських”), що утворюють ієрархічну структуру вузлів. Вузли можуть бути елементами будь-якого типу (літерами, рядками, числами).



Піддерево, корінь якого знаходиться в лівому (правому) нащадку вершини, називається ***лівим*** (***правим***) ***піддеревом*** цієї вершини.

Висота вузла дерева - це довжина самого довгого шляху з цього вузла до будь-якого листа.

Висота дерева співпадає з висотою кореня.

Глибина вузла – це довжина шляху від кореня до цього вузла.

Степінь вузла – це кількість дуг, що з нього виходить.

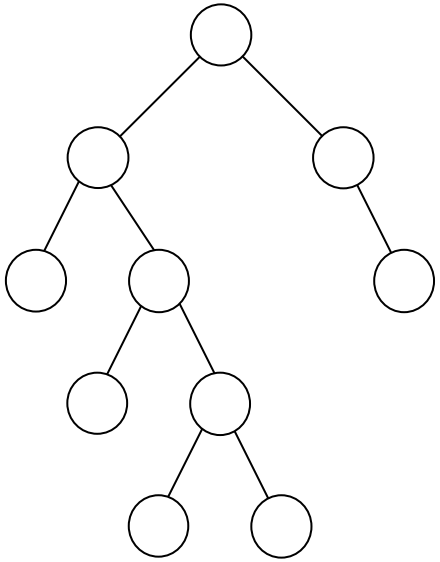
Степінь дерева дорівнює максимальному степеню вузла, що входить у дерево.

Листя в дереві - це вузли, що мають степінь нуль.

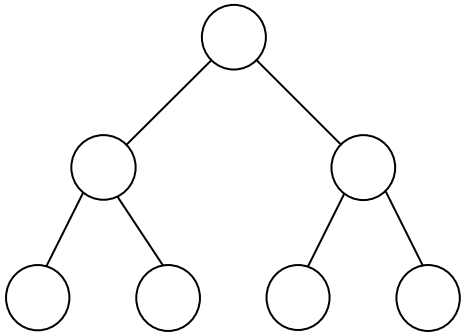
Бінарне дерево – це дерево степінь якого дорівнює два .

Дерева, степінь яких більше двох, називаються **розгалуженими**.

Повне бінарне дерево - це дерево для якого на всіх рівнях менше чим *n* вузли мають степінь 2, а на рівні *n* – степінь 0.



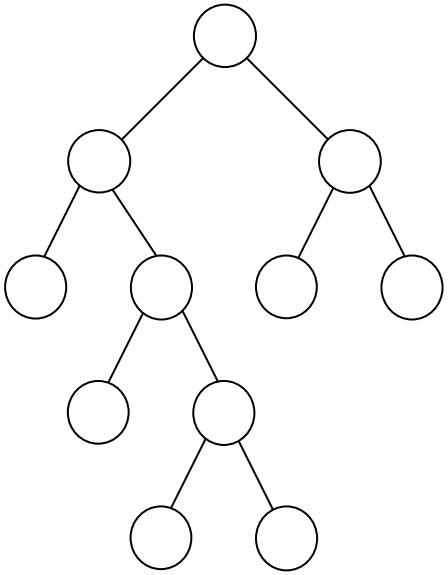
а) неповне бінарне дерево



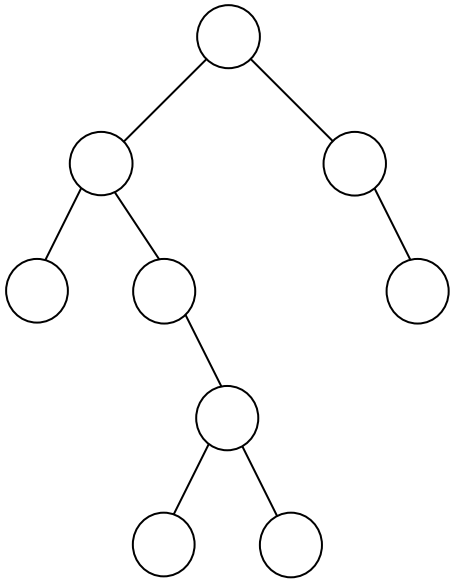
б) повне бінарне дерево

Строго бінарне дерево складається тільки з вузлів, що мають степінь 2 або 0.

Нестрого бінарне дерево містить вузли зі степенем 1.



а) строго бінарне дерево



б) нестрого бінарне дерево

§2. Бінарні дерева пошуку

Бінарне дерево пошуку (binary search tree – BST) – це бінарне дерево в якому:

- Кожен вузол має не більше двох нащадків;
- Кожен вузол має ключ (key) і значення (value);
- Ключі всіх вузлів лівого піддерева менші значення ключа батьківського вузла;
- Ключі всіх вузлів правого піддерева більші значення ключа батьківського вузла.

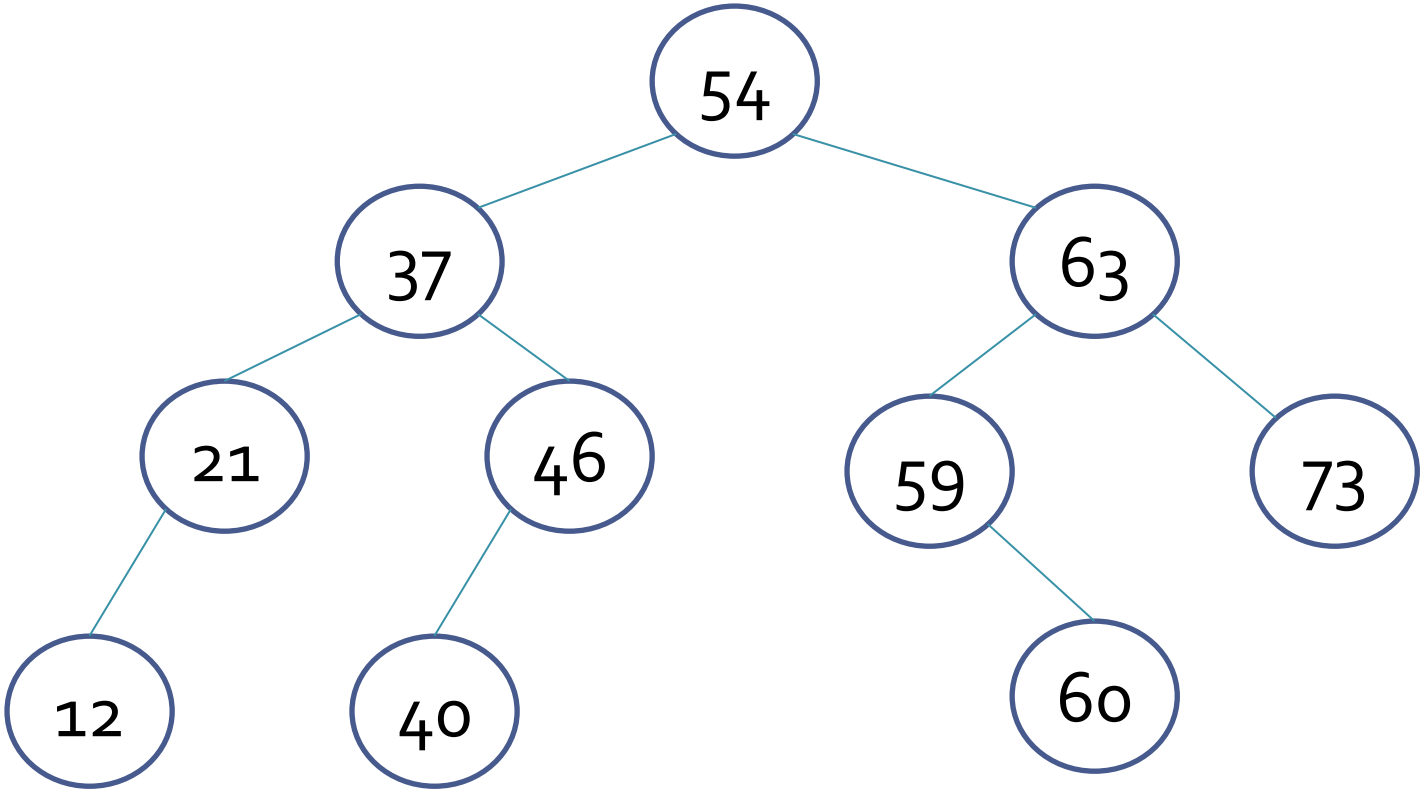
Використовуються для реалізації словників та множин.

Алгоритм вставки елемента

1. Починаємо з кореня.
2. Якщо елемент < об’єкта в вершині, переходимо до лівого сина.
3. Якщо елемент > об’єкта в вершині, переходимо до правого сина.
4. Повторюємо кроки 2 і 3, доки не досягнемо вершини, яка не визначена.
5. Якщо досягнута вершина не визначена, то визначаємо вершину і вставляємо елемент.

Приклад. Побудувати дерево пошуку:

54, 37, 63, 21, 46, 73, 59, 12, 40, 60.



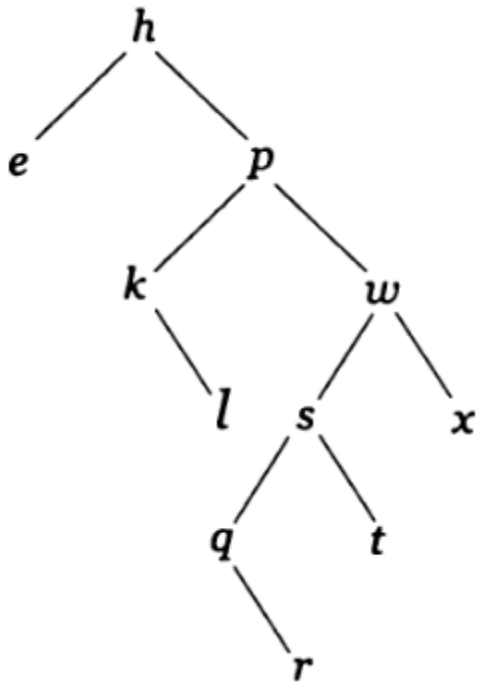
Алгоритм пошуку елемента

1. Починаємо з кореня.
2. Якщо елемент < об’єкта в вершині, переходимо до лівого сина.
3. Якщо елемент > об’єкта в вершині, переходимо до правого сина.
4. Якщо елемент = об’єкту в вершині, то елемент знайдено; виконуємо відповідні дії і виходимо.
5. Повторюємо кроки 2, 3 і 4 доки не досягнемо вершини, яка не визначена.
6. Якщо досягнута вершина не визначена і в дереві немає шуканого елемента, то виконуємо відповідні дії і виходимо.

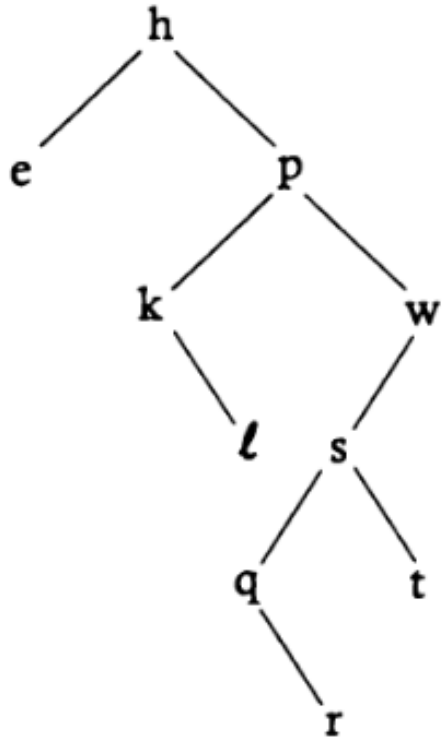
Алгоритм видалення елемента

1. Якщо вершина v_0 не має синів, просто видаляємо її.
2. Якщо вершина v_0 має одного сина, видаляємо v_0 і заміняємо її сином.
3. Якщо v_0 має двох синів, знаходимо правого сина v_1 вершини v_0, а потім знаходимо лівого сина вершини v_1 (якщо він існує). Продовжуємо вибирати лівих синів кожної знайденої вершини, доки не знайдеться така вершина v, у якої не буде лівого сина. Замінимо v_0 на v і зробимо правого сина вершини v лівим сином батька вершини v.

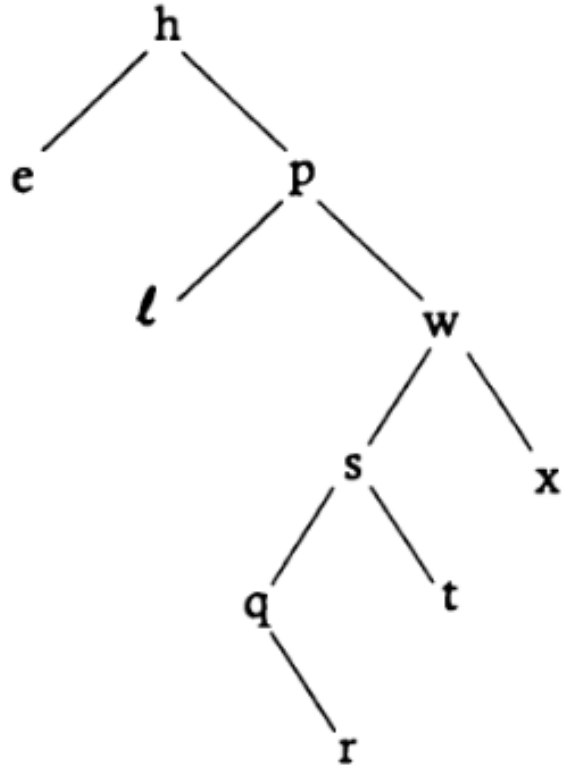
Задане дерево



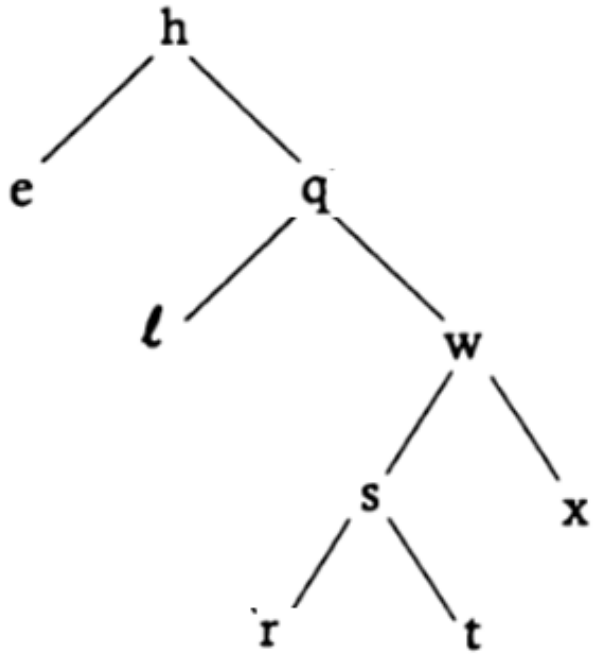
Дерево, після видалення вершини ***x***



Дерево, після видалення вершини ***k***



Якщо v_0 має двох синів, знаходимо правого сина v_1 вершини v_0, а потім знаходимо лівого сина вершини v_1 (якщо він існує). Продовжуємо вибирати лівих синів кожної знайденої вершини, доки не знайдеться така вершина v, у якої не буде лівого сина. Замінимо v_0 на v і зробимо правого сина вершини v лівим сином батька вершини v.



Збалансоване дерево пошуку – це дерево пошуку, в якому число вершин в його лівих і правих піддеревах відрізняються не більше ніж на 1.

Збалансовані дерева пошуку:

- Червоно-чорні дерева
- АВЛ – дерева
- 2-3-дерева
- B-дерева
- …

§3. Обхід дерев

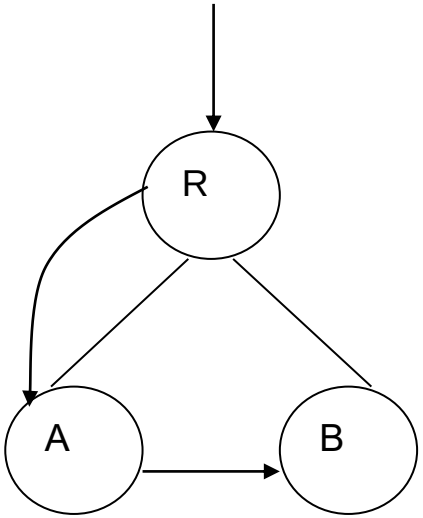
Під **обходом** бінарного дерева розуміють визначений порядок проходження всіх вершин дерева. Розрізняють: прямий, зворотній та симетричний порядки обходу.

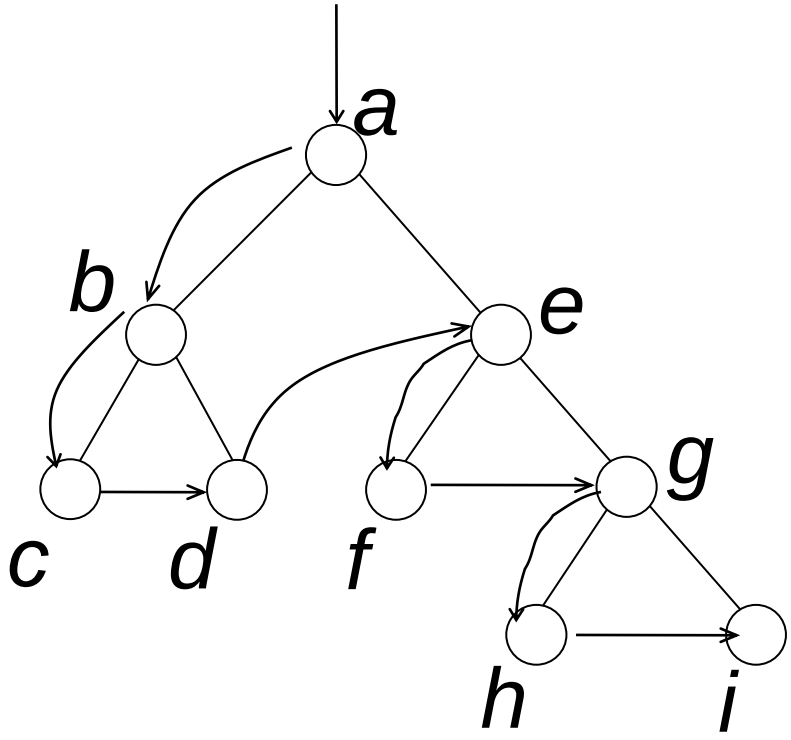
Прямий порядок обходу*:*

1.почати з кореня R

2.пройти в прямому порядку ліве піддерево A

3.пройти в прямому порядку праве піддерево B

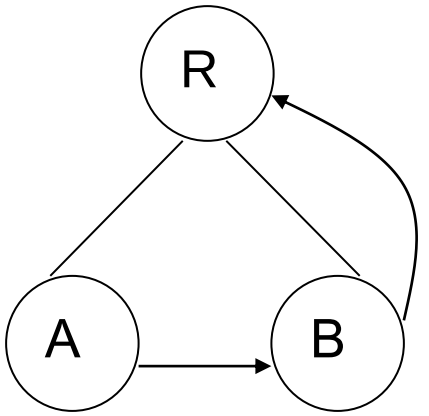


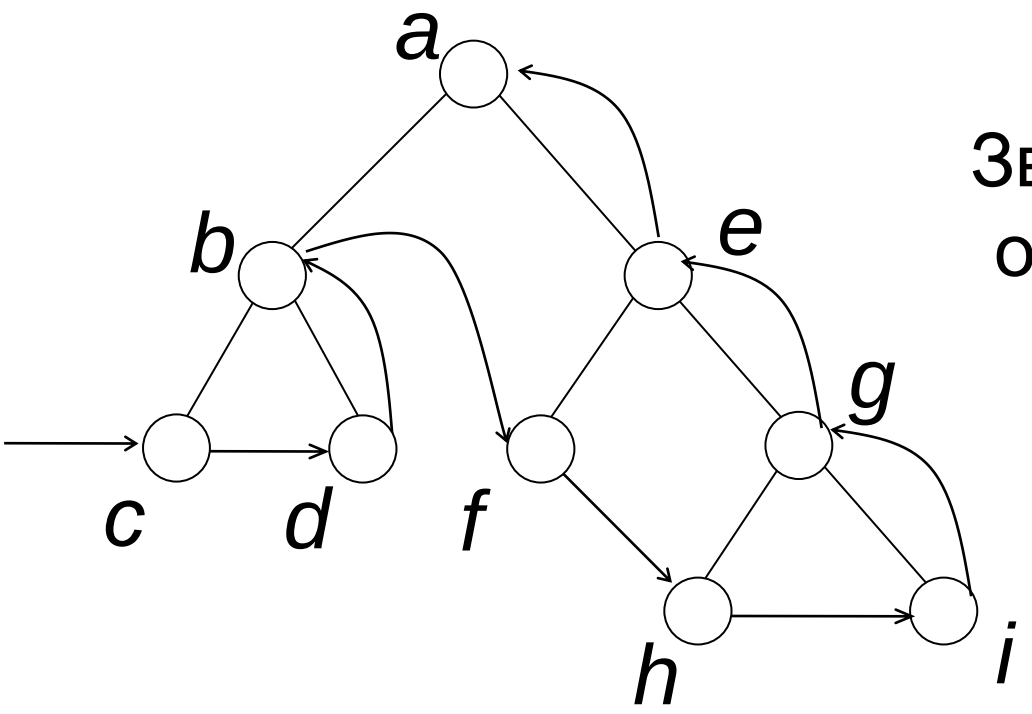


Прямий порядок обходу бінарного дерева: *a b c d e f g h i*

Обхід дерева в *зворотньому* порядку:

- пройти в зворотному порядку ліве піддерево A
- пройти в зворотному порядку праве піддерево B
- потрапити в корінь R



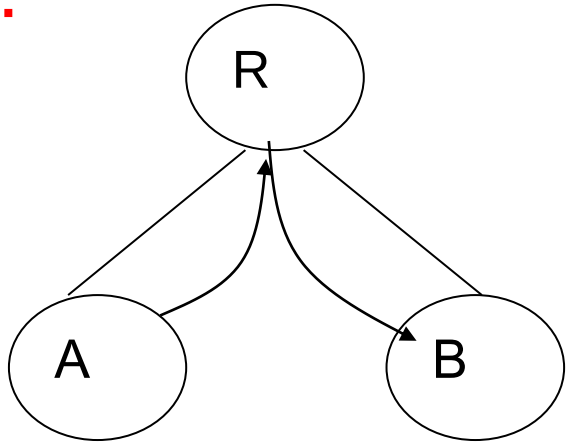


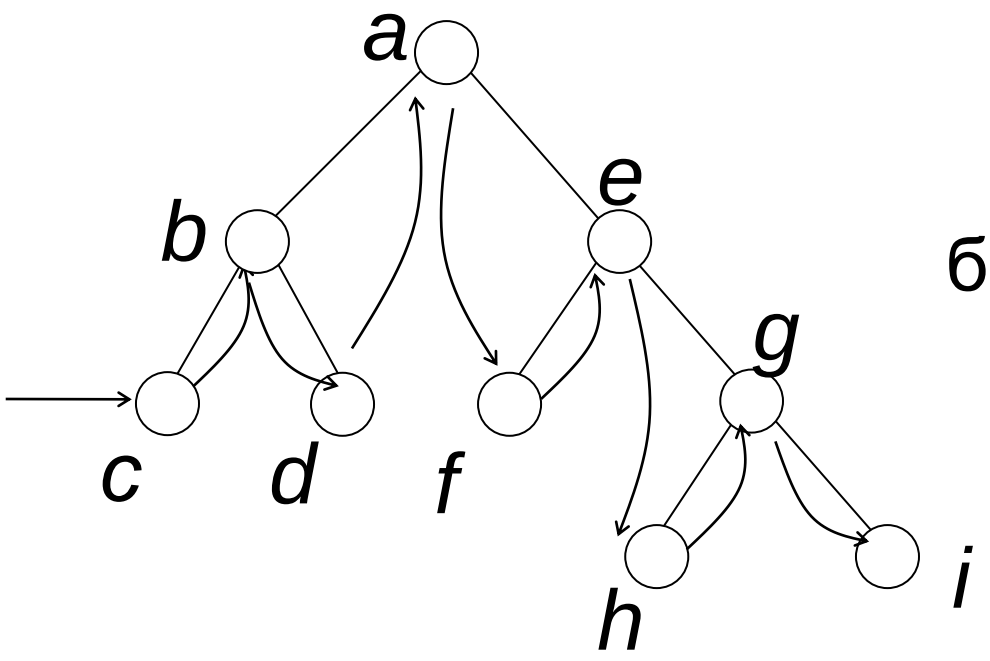
Зворотній порядок обходу бінарного дерева:

c d b f h i g e a

Симетричний порядок обходу бінарного дерева:

- пройти в симетричному порядку ліве піддерево A
- потрапити в корінь R
- пройти в симетричному порядку праве піддерево B





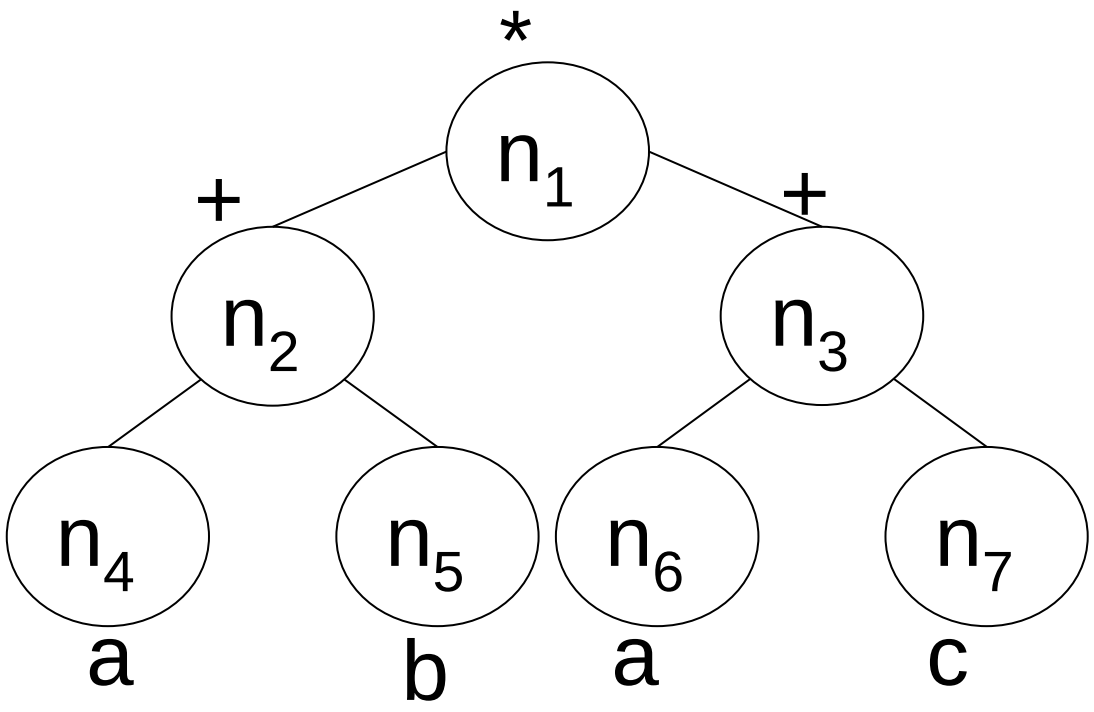
Симетричний порядок обходу бінарного дерева:
c b d a f e h g i

§4. Дерева виразів

Часто при обході дерев складається список не імен вузлів, а їх міток (label) – значень, які зберігаються у вузлі. Такі дерева називаються ***дерева з мітками***. Проводять наступну аналогію: дерево – список, вузол – позиція, мітка – елемент.

За допомогою дерев можна представляти довільні арифметичні вирази. Кожному листові в такому дереві відповідає операнд, а кожному батьківському вузлу – операція. У загальному випадку дерево при цьому може виявитись не бінарним.

Приклад: На рисунку наведено дерево з мітками, що представляє арифметичний вираз (a+b)*(a+c), де n_1, n_2 …n_7 – імена вузлів, а мітки представлені поруч з відповідними вузлами.



- При прямому впорядкуванні міток отримуємо ***префіксну*** форму виразів, де оператор передує і левому і правому операндам. Префіксний вираз: *+ab+ac.
- Зворотнє впорядкування міток дерева виразів дає ***постфіксне*** (або польське) представлення виразів, при якому оператор іде після лівого і правого операндів. Постфіксна форма: ab+ac+*.
- При симетричному обході дерев виразів отримаємо так звану ***інфіксну*** форму виразу, яка співпадає зі звичайною стандартною формою запису виразу, але не використовує дужок.

  Інфіксний вираз: a+b*a+c.

Лекція 8.

Алгоритми пошуку остовних дерев

Що таке обхід графа?

Простими словами, обхід графа — це перехід від одніє його вершини до іншої в пошуку властивостей зв`язків цих вершин. Зв`язки (лінії, що з`єднують вершини) називаються напрямками, шляхами, гранями або ребрами графа. Вершини графа також називаються вузлами.

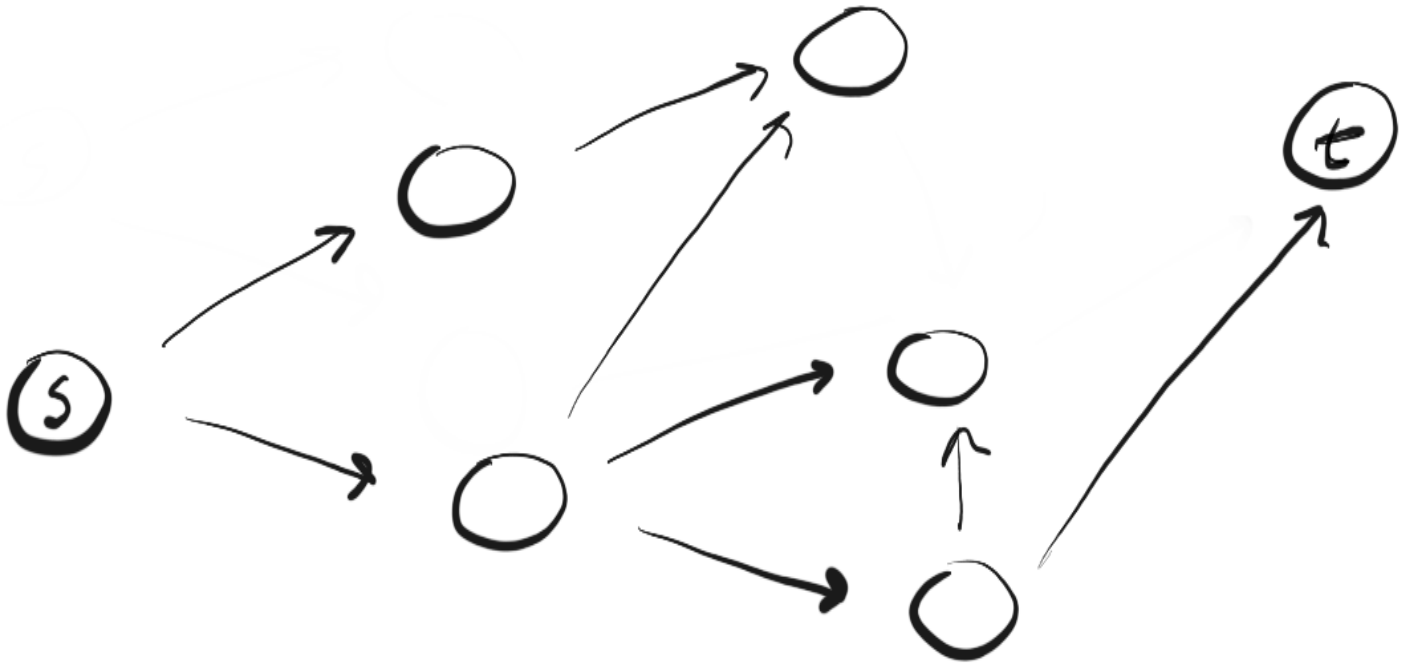
Двома основними алгоритмами обходу графа є пошук в глибину (Depth-FirstSearch, DFS) і пошук в ширину (Breadth-FirstSearch, BFS).

Не дивлячись на те, що обидва алгоритми використовуються для обходу графа, вони мають деякі відмінності. Почнемо з DFS.

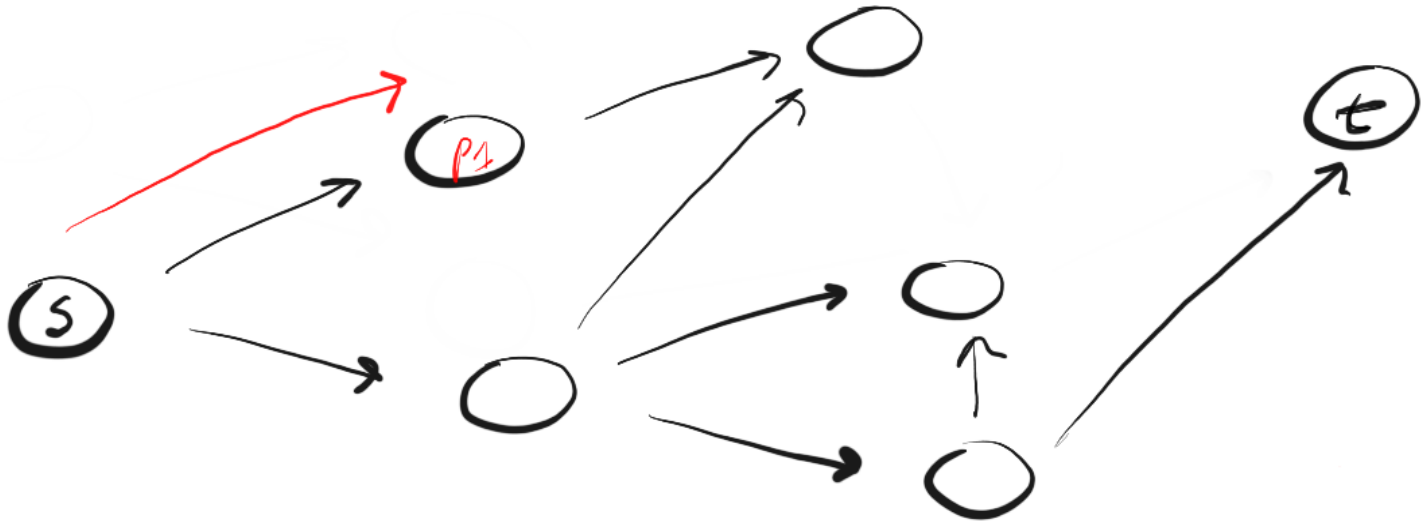
Пошук в глибину

DFS слідує концепції «поринай глибже, головою вперед» («godeep, headfirst»). Ідея полягає в тому, що ми рухаємося від початкової вершини (точки, місця) в заданому напрямку (за заданим шляхом) до тих пір, поки не досягнемо кінця шляху або пункту призначення (шуканої вершини). Якщо ми досягли кінця шляху, але він не є пунктом призначення, то ми повертаємося назад (до точки розгалуження шляхів) і йдемо по іншому маршруту.

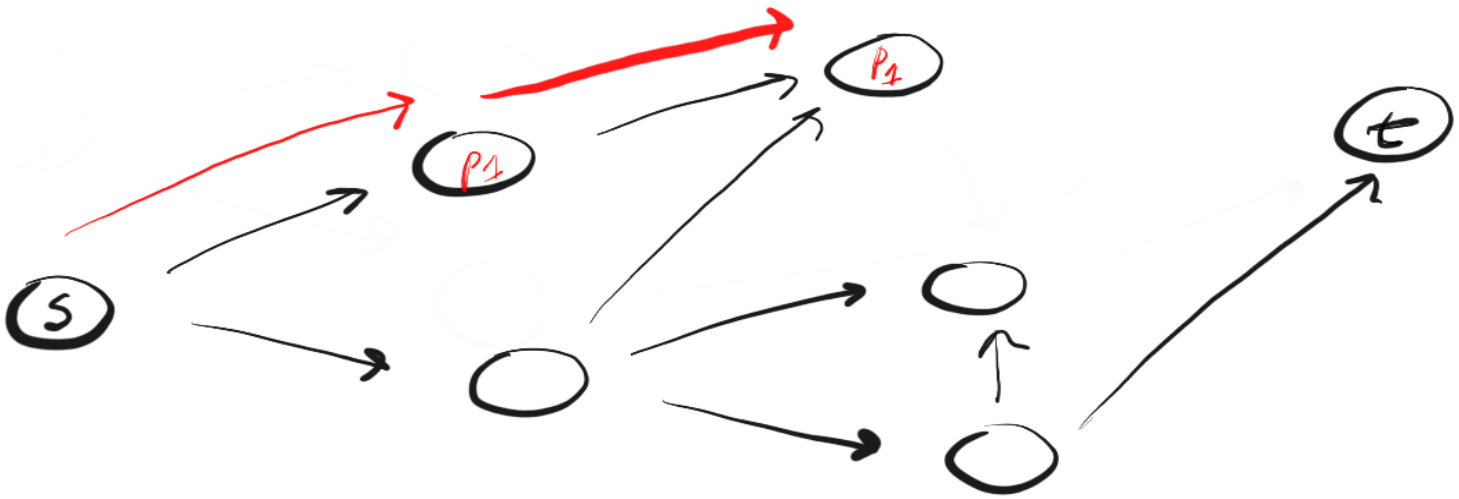
Давайте розглянемо приклад. Нехай, в нас є орієнтований граф, який має наступний вигляд:



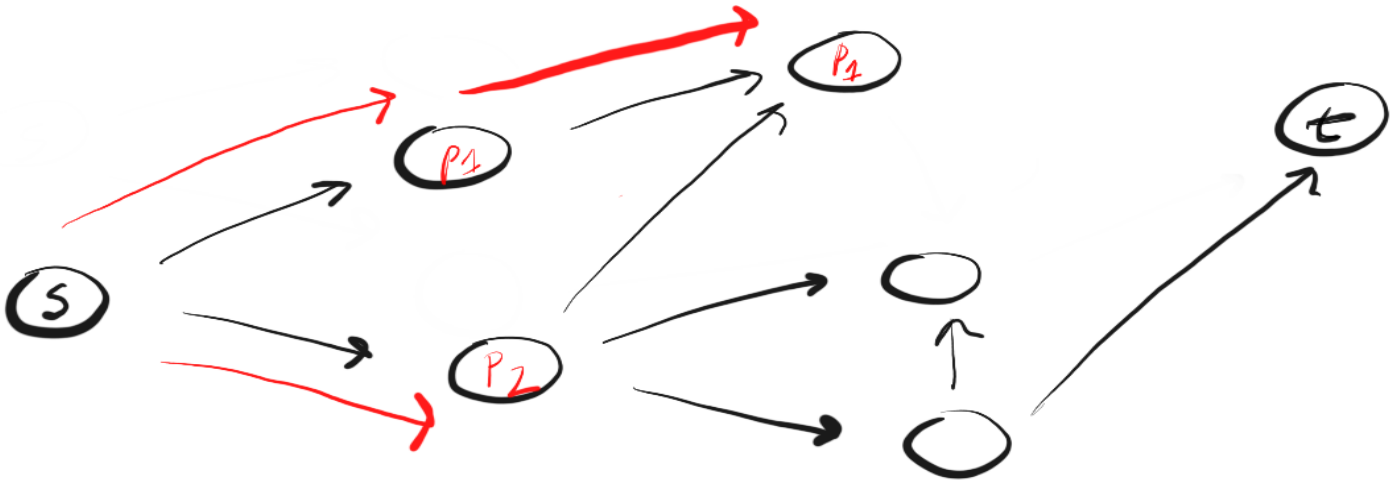
Ми знаходимося в точці «s» і нам потрібно знайти вершину «t». Застосовуючи DFS, ми досліджуємо один з можливих шляхів, рухаємося по ньому до кінця і, якщо не знайшли t, повертаємося і досліджуємо інший шлях. Ось як виглядає процес:



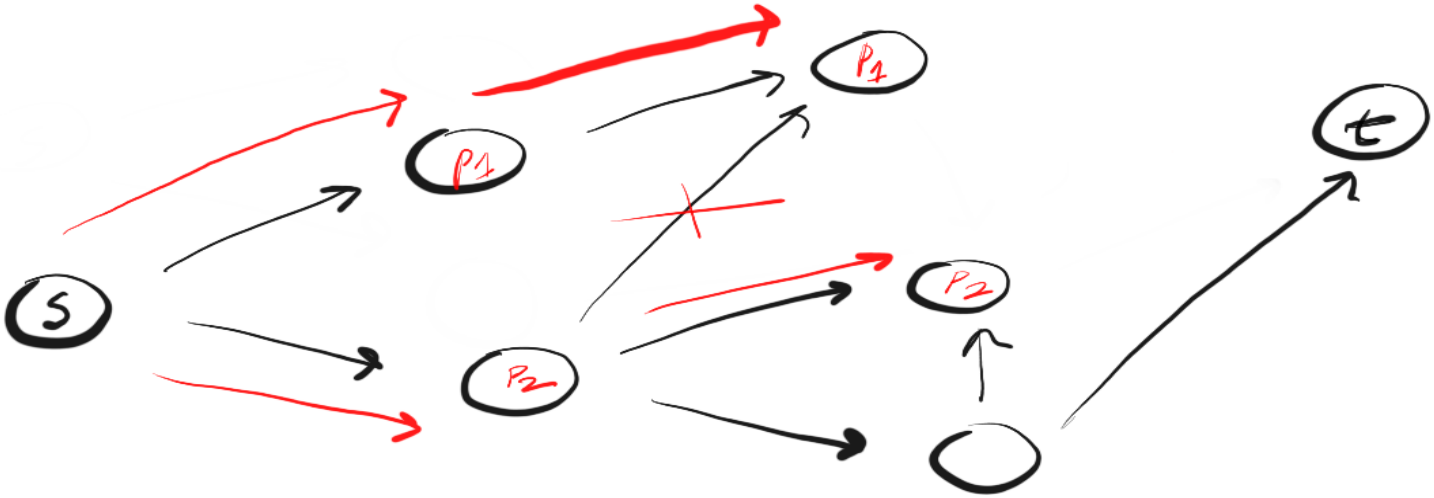
Тут ми рухаємося по маршруту (p1) до найближчої вершини і бачимо, що це не кінець шляху. Тому ми переходимо до настуної вершини.



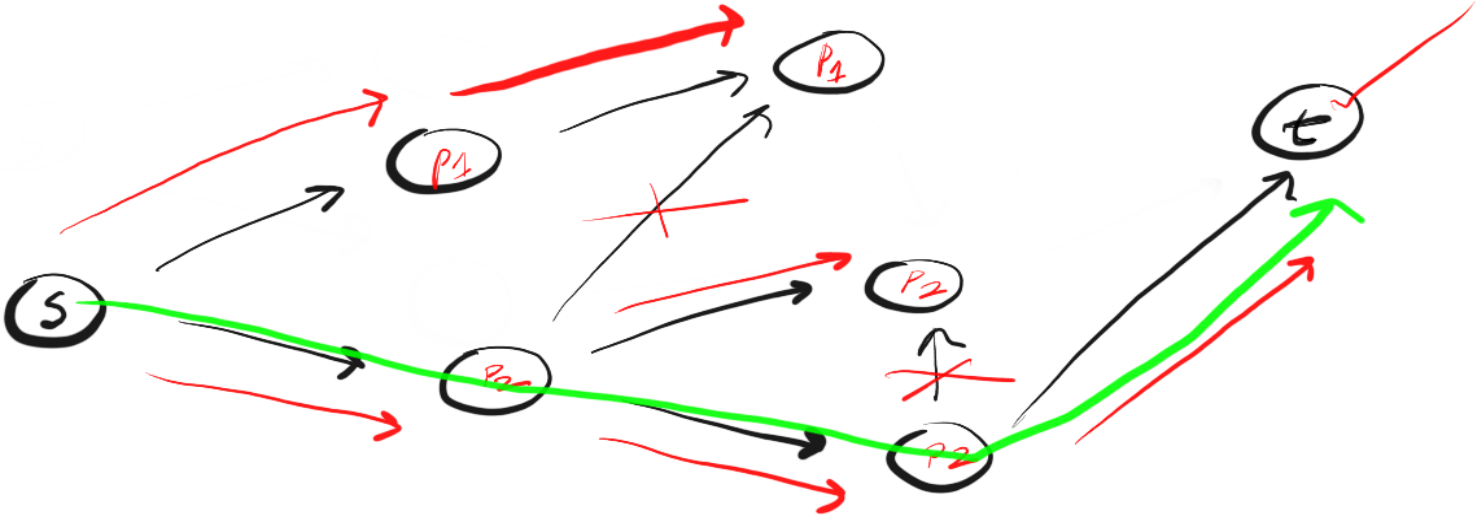
Ми досягли кінця p1, але не знайшли t, тому повертаємося в s і рухаємося по іншому маршруту.



Досягнувши найближчої доточки «s» вершини шляху «p2» ми бачимо три можливих напрямки для подальшого руху. Оскільки вершину, де закінчувався перший напрямок, ми вже відвідували, то рухаємося по другому.



Ми знову досягли кінця шляху, але не знайшли t, тому повертаємося назад. Слідуємо за третім шляхом і, нарешті, досягаємо шуканої вершини «t».



Так працює DFS. Рухаємося за заданим шяхом до кінця. Якщо кінець шляху — це шукана вершина, ми закінчили. Якщо ні, повертаємося назад і рухаємося іншим шляхом, доти, доки не дослідимо всі варіанти.

Ми слідуємо за цим алгоритмом, застосовуючи його до кожної відвідуваної вершини.

Необхідність багаторазового повторення процедури вказує на необхідність застосування рекурсії для реалізації алгоритму.

Вот JavaScript-код:

```
// за умови, що ми маємо справу з суміжним списком
// наприклад, таким: adj = {A: [B,C], B:[D,F], ... }
Function dfs(adj, v, t) {
        // adj - суміжний список
        // v – відвіданий вузол (вершина)
        // t - пункт призначення
        // це загальні випадки
        // або досягли пункту призначення, або вже відвідували вузол
        if(v === t) return true
        if(v.visited) return false
        // помічаємо вузол як відвіданий
        v.visited = true
        // досліджуємо всіх сусідів (найближчі сусідні вершини) v
        for(let neighbor of adj[v]) {
                // якщосусід не відвідувався
                if(!neighbor.visited) {i
                        // рухаємося шляхом і перевіряємо, чи не досягли ми пункту
призначення
                        let reached = dfs(adj, neighbor, t)
                        // повертаємо true, якщодосягли
                        if(reached) return true
                }
        }
        // якщо від v до t дістатися неможливо
        return false
}
```

Примітка: цей спеціальний DFS-алгоритм дозволяє перевірити, чи можливо дістатися з одного місця в інше. DFS може застосовуватися з різною метою. Від цієї мети залежить те, як буде виглядати сам алгоритм. Тим не менше, загальна концепція виглядає саме так.

Аналіз DFS

Давайте проаналізуємо цей алгоритм. Оскільки ми обходимо кожного «сусіда» кожного вузла, ігноруючи тих, яких відвідували раніше ми маємо час виконання, що дорівнює O(V + E).

Коротке пояснення того, що означає V+E:

V — загальна кількість вершин. E — загальна кількість граней (ребер).

Може здатися, що правильніше застосувати V*E, однак давайте подумаємо, що означає V*E.

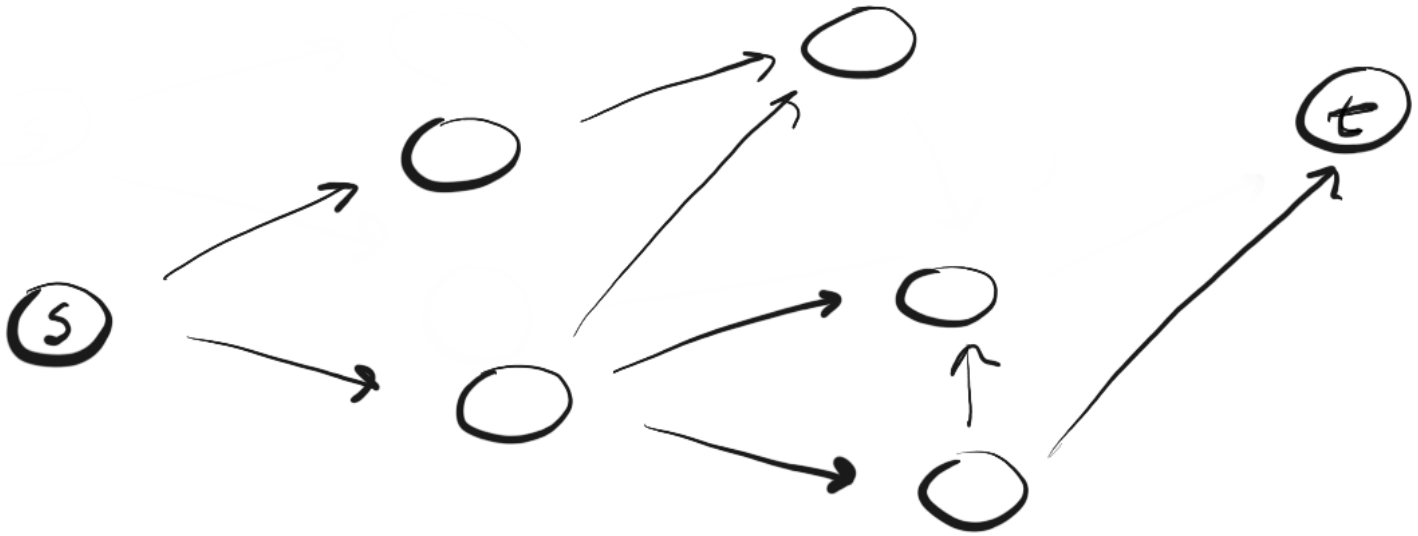
V*E означає,що стосовно кожної вершини, ми повинні дослідити всі грані графа безвідносно приналежності цих граней конкретній вершині.

З іншого боку, V+E означає, що для кожної вершини ми оцінюємо лише грані, що примикають до неї. Повертаючись до прикладу, кожна вершина має визначену кількість граней і, в гіршому випадку, ми обійдемо всі вершини (O(V)) і дослідимо всі грані (O(E)). Ми маємо V вершин і E граней, тому отримаємо V+E.

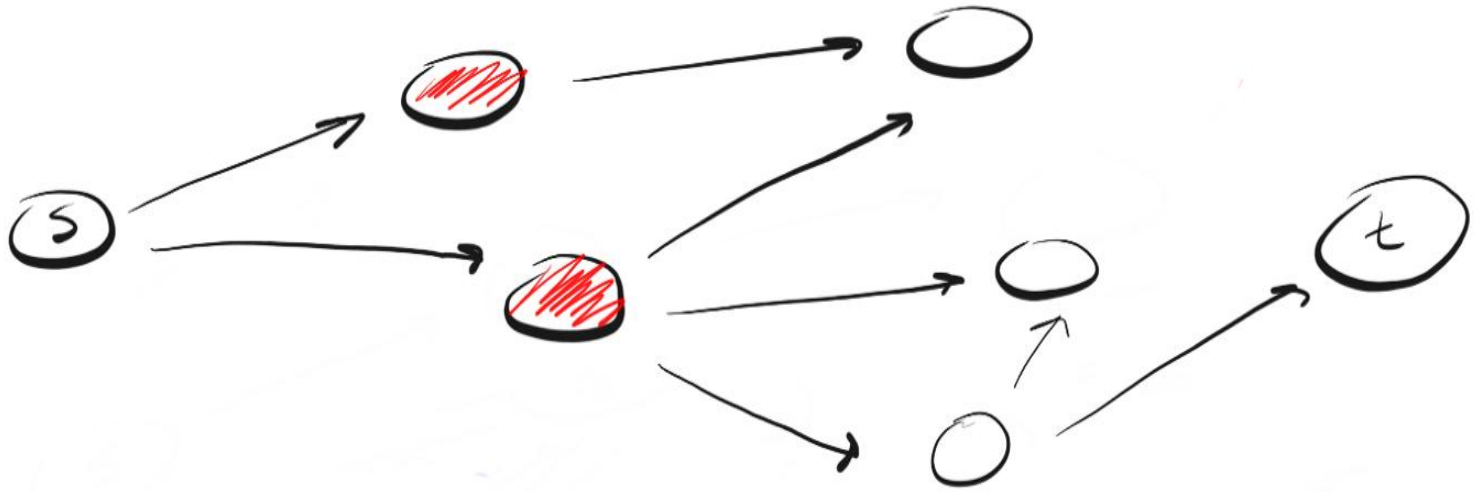
Далі, оскільки ми застосували рекурсію для обходу кожної вершини, це означає, що застосовується стек (безкінечна рекурсія приводить до помилки переповнення).

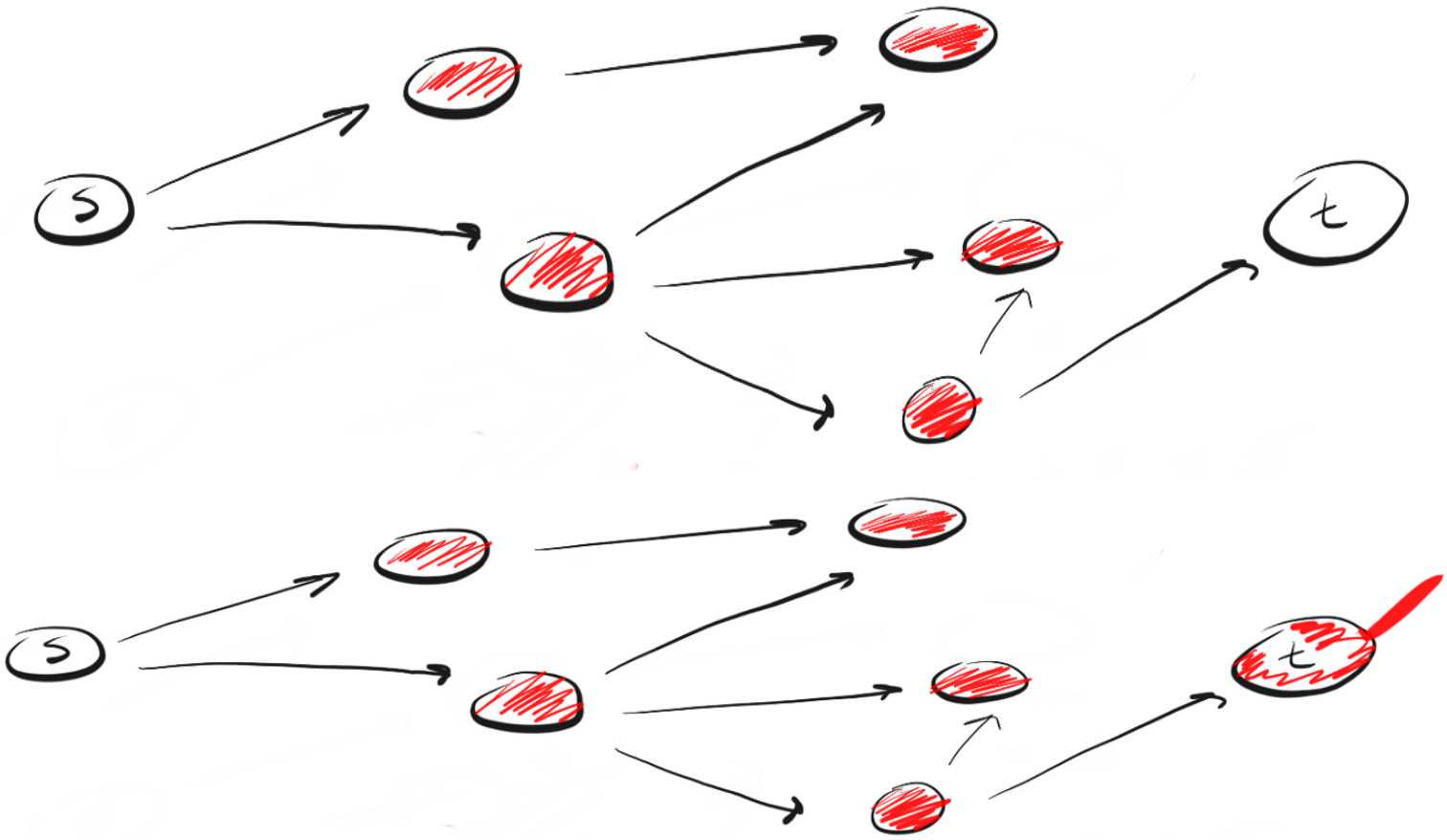
Пошук в ширину

BFS слідує концепції «розширюйся, піднімаючись на висоту пташиного польоту» («go wide, bird’se ye-view»). Замість того, щоб рухатися заданим шляхом до кінцяа, BFS передбачає рух вперед по одному сусіду за раз. Це означає наступне:



Замість слідування шляхом, BFS передбачає відвідування найближчих до s сусідів за один крок, потім відвідування сусідів і так далі до тих пір, поки не буде досягнута t.




s
t
s
t

Чим DFS відрізняється від BFS? Мені здається що, DFS йде напролом, а BFS не поспішає, а досліджує все в межах одного кроку.

Далі виникає питання: як дізнатися, яких сусідів слід відвідати першими?

Для цього ми можемо використати концепцію «першим зайшов, першим вийшов» (first-in-first-out, FIFO) з черги (queue). Ми заносимо в чергу спочатку найближчу до нас вершину, потім її невівіданих сусідів, і продовжуємо цей процес, доки черги не залишиться або доки ми не знайдемо шукану вершину.

Ось код:

```
// за умовою, що ми маємо справу з суміжним списком
// наприклад, таким: adj = {A:[B,C,D], B:[E,F], ... }
functionbfs(adj, s, t) {
	// adj - суміжний список
	// s - початкова вершина
	// t - пункт призначення
	// ініціалізуємо чергу
	letqueue = []
	// додаємо s в чергу
	queue.push(s)
	// помічаємо s як відвідану вершину, щоб не повторити додавання в чергу
	s.visited = true
	while(queue.length> 0) {
		// видаляємо перший (верхній) елемент з черги
		let v = queue.shift()
		// abj[v] - сусіди v
		for(let neighbor of adj[v]) {
			// якщо сусід не відвідувався
			if(!neighbor.visited) {
				// додаємо його в чергу
				queue.push(neighbor)
				// помічаємо вершину як відвідану
				neighbor.visited = true
				// якщо сусід є пунктом призначення, ми перемогли
				if(neighbor === t) return true
			}
		}
	}
	// якщо t не знайдена, значить пункта призначення досягти неможливо
	returnfalse
}
```

Аналіз BFS

Може здатися, що BFSпрацює повільніше. Однак,якщо уважно придивитися до візуалізацій, можно побачити, що вони мають однаковий час виконання.

Черга передбачає обробку кожної вершини перед досягненням пункту приазначення. Це означає, що, в гіршому випадку, BFS досліджує всі вершин і грані.

Не дивлячись на те, що BFS може здаватися повільнішим, але він скоріший, оскільки при роботі з великими графами виявляється, що DFS витрачає багато часу на слідування шляхами, які в кінцевому рахунку виявляються хибними. BFS часто застосовують для знаходження найкоротшого шляху між двома вершинами.

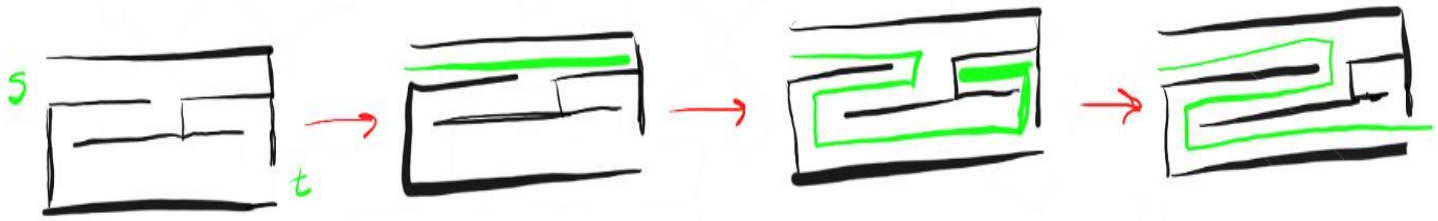
Таким чином, виконання BFS також складає O(V + E), а оскільки ми застосовуємо чергу, що вміщує всі вершини, його просторова складність складає O(V).

Аналоги з реального життя

Якщо наводити приклади з реального життя, то ось як я представляю собі роботу DFS і BFS.

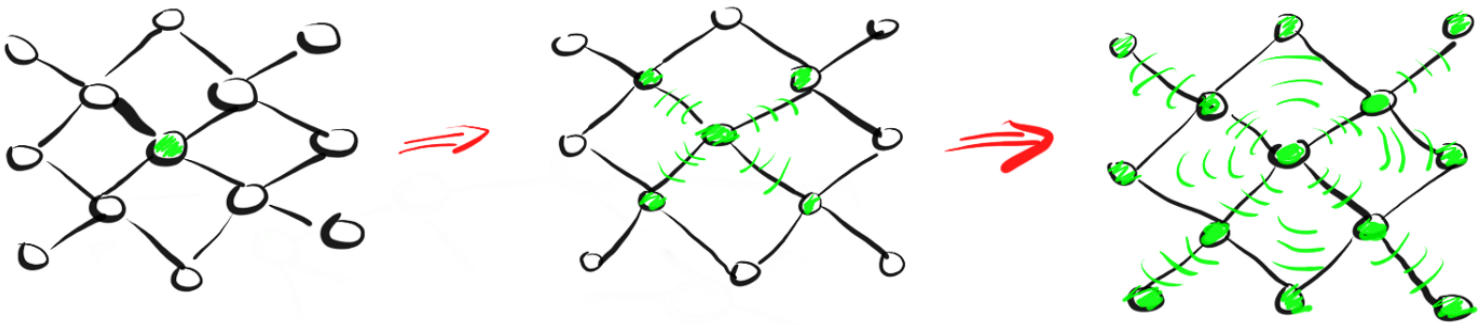
Коли я думаю про DFS, то представляю собі мишу в лабіринті в пошуку їжі. Для того, щоб потрапити до їжі миша змушена багато разів потрапляти в тупик, повертатися і рухатися в іншому напрямку, і так докивона не знайде вихід з лабіринту або їжу.

Спрощена версія виглядає так:



В свою чергу, коли я думаю про BFS, то представляю собі кола на воді. Падіння каменю в воду приводить до утворення кіл по всіх напрямках від центру.

Спрощена версія виглядає так:



Висновки

•Пошук в глибину і пошук в ширину використовують для обходу графа.

•DFS рухається по гранях (ребрах) туди і назад, а BFS розповсюджується по сусідах в пошуку призначення.

•DFS використовує стек, а BFS — чергу.

•Час виконання обох складає O(V + E), а просторова складність — O(V).

•Дані алгоритми мають різну філософію, але однаково важливі для роботи з графами.

Алгоритм пошуку максимальної кількості остовних дерев

Цей алгоритм застосовується для будь-якого зв'язного графа G(V,E) з поміченими вершинами, в якого необхідно знайти всі можливі остовні дерева.

Кроки алгоритму пошуку максимальної кількості остовних дерев

•Знайти степені вершин цього зв'язного графа;

•Побудувати матрицю степенів вершин зв'язного графа;

•Побудувати матрицю суміжності зв'язного графа;

•Знайти різницю між матрицями степенів і суміжності зв'язного графа G(V,E);

•Знайти будь-яке з алгебраїчних доповнень отриманої матриці різниці на кроці 4 алгоритму, значення якого є кількісним значенням остовних дерев для графа G(V,E).

Приклад. Використовуючи розглянутий алгоритм, знайти максимальну кількість остовних дерев для зв’язного графа G(V,E) , наведеного на рисунку: