Алгоритми та складність

II семестр

Лекція 1

Структура курсу

14 лекційних занять

14 практичних занять

- На лекціях – 2 підсумкових модулі.
- В кінці – **іспит**!
- Бали: 30 (лекції) + 30 (практика).
- Допуск до іспиту: набір 36 балів в семестрі (за умови виконання лабораторних мінімум на 60% – це 18 балів).

Папка з прочитаними лекціями (незмінна)

https://drive.google.com/drive/folders/1xGNrjkSAWTiDLAU_ekGRMv3PK4cFxbg1

Структури даних

- *Динамічна множина* – може змінюватися в ході виконання алгоритму.
- Розглядатимемо скінченні динамічні множини.
- Якщо елементи складні, одне з їх полів визначатиметься як *ключове*.
- Деякі поля можуть бути доступними для маніпуляцій під час виконання операцій над множиною.
- Складні елементи можуть містити в інших полях *супутні дані*, що не використовуються реалізацією множини.
- Якщо всі ключі різні, динамічна множина може бути представлена набором ключових значень.
- Іноді ключі динамічної множини є членами цілком впорядкованої множини.

Операції на динамічних множинах

Типи операцій

Запити

Повертають інформацію про множину

Модифікуючі операції

Змінюють множину

Типові операції на динамічних множинах

В кожному конкретному випадку потрібна реалізація лише частини з них. Наприклад, для словника необхідні вставка, видалення та перевірка належності елемента множині.

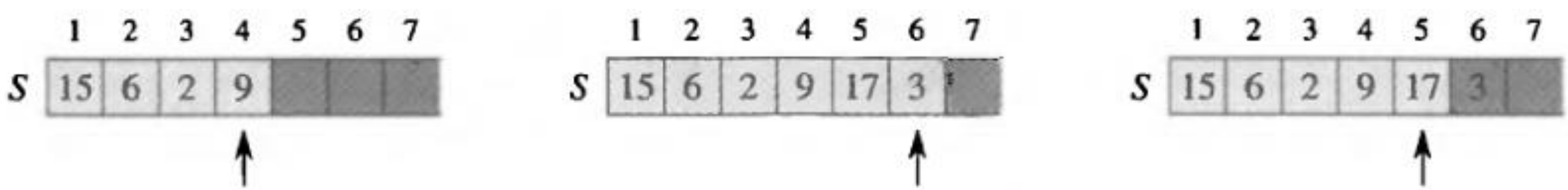
- SEARCH(S,k): запит повертає вказівник на елемент x заданої множини S, для якого key[x]=k або NIL, якщо такого елемента в S немає.
- INSERT(S,x): модифікуюча операція, що поповнює множину S одним елементом, на який вказує x; вважають, що всі поля x, необхідні для реалізації множини вже попередньо ініціалізовані.
- DELETE(S,x): модифікуюча операція, що видаляє з множини S елемент, на який вказує x.

Типові операції на динамічних множинах

- MINIMUM(S): запит до цілком впорядкованої множини S, що повертає вказівник на елемент з найменшим ключем.
- MAXIMUM(S): запит до цілком впорядкованої множини S, що повертає вказівник на елемент з найбільшим ключем.
- SUCCESSOR(S,x): запит до цілком впорядкованої множини S, що повертає вказівник на елемент множини, ключ якого є найближчим більшим сусідом ключа елементу x; якщо x – максимальний елемент, повертається NIL.
- PREDECESSOR(S,x): запит до цілком впорядкованої множини S, що повертає вказівник на елемент множини, ключ якого є найближчим меншим сусідом ключа елементу x; якщо x – мінімальний елемент, повертається NIL.

Елементарні структури даних. Стеки

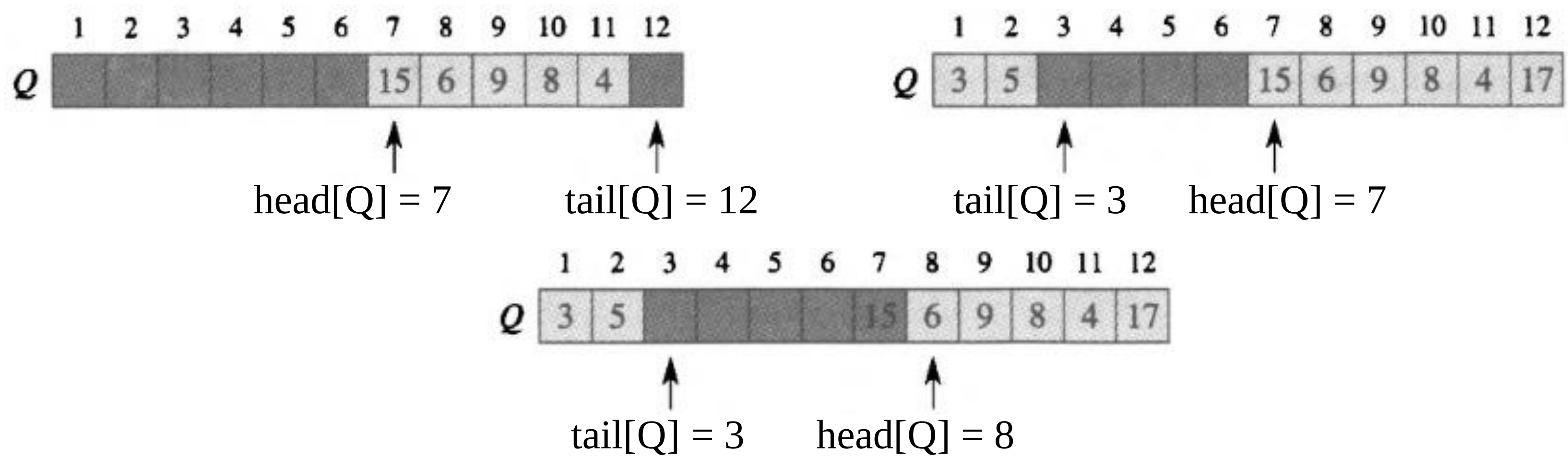
- Стратегія «останній прийшов – перший вийшов» (LIFO)
- PUSH: поміщення елементу в вершину стеку
- POP: зняття елементу з вершини стеку
- Приклад реалізації стека з не більше ніж n елементів: масив S[1..n], через *top*[*S*] визначається індекс верхнього елемента
- Додатково вводяться операції-запити перевірки порожності/заповненості стека
- Всі операції виконуються за час O(1)



top[S] = 4 top[S] = 6 top[S] = 5

Елементарні структури даних. Черги

- Стратегія «перший прийшов – перший вийшов» (FIFO)
- ENQUEUE: поміщення у хвіст черги
- DEQUEUE : взяття елементу з голови черги
- Приклад реалізації черги з не більше ніж (n–1) елементів: циклічний масив Q[1..n], через *head*[Q] визначається індекс голови, через *tail*[Q] – місце додавання нового елемента
- Всі операції виконуються за час O(1)



Елементарні структури даних. Деки

- Дек (deque) – це черга з двостороннім доступом.
- Допустимі операції вставки і видалення елементів з обох кінців.
- Англійська назва deque приховує гру слів: написання "d-e-que" як скорочення "double-ended queue" і вимова як у "deck" через схожість з колодою карт, звідки можна здавати як зверху, так і знизу.
- Так само може бути реалізований на циклічному масиві.
- Всі операції виконуються за час O(1).

Елементарні структури даних. Зв’язані списки

- *Зв’язаний список* – структура даних, в якій елементи розташовані в лінійному порядку, що визначається вказівниками на кожен об’єкт.
- Однобічно зв’язаний список: поле-ключ *key* та вказівник на наступний елемент *next*.
- Двобічно зв’язаний список: поле-ключ *key* та вказівники на наступний (*next*) та попередній (*prev)* елементи.
- Кільцевий список: перший та останній елементи зв’язані: *next* останнього елемента вказує на голову, а *prev* першого на хвіст.
- Відсортований список: лінійний порядок списку відповідає лінійному порядку його ключів: мінімальний елемент міститься в голові, максимальний – в хвості.

Елементарні структури даних. Зв'язані списки

Якщо не вказано інакше, списки вважатимемо двозв'язними невідсортованими.

- Пошук в списку L за ключем

LIST_SEARCH(L, k)

$x \leftarrow head[L]$

while $x \neq$ NIL и $key[x] \neq k$

    do $x \leftarrow next[x]$

return x

В найгіршому випадку час Θ(n) для списку в n елементів.

Елементарні структури даних. Зв’язані списки

- Вставка в голову списку L

```
LIST_INSERT(L, x)
next[x] ← head[L]
if head[L] ≠ NIL
    then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL
```

Час роботи O(1).

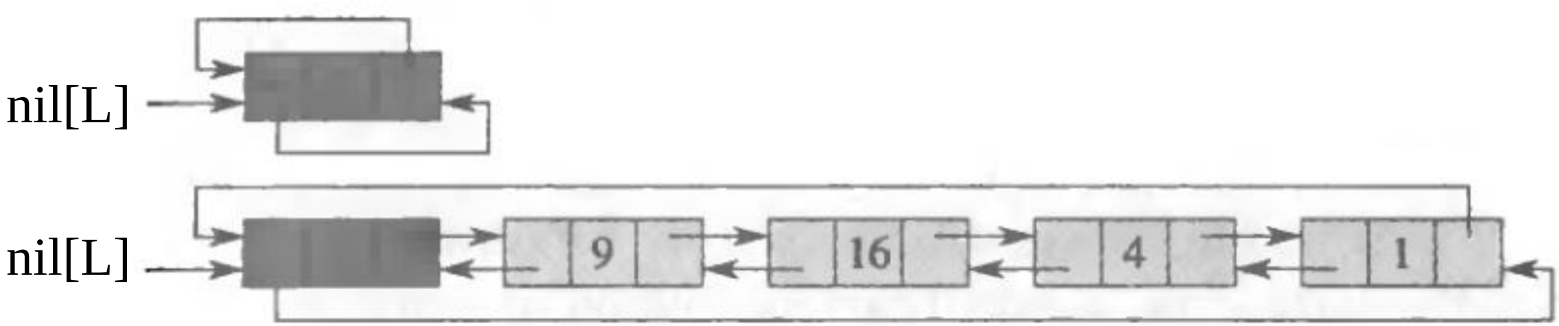
- Видалення за вказівником

```
LIST_DELETE(L, x)
if prev[x] ≠ NIL
    then next[prev[x]] ← next[x]
    else head[L] ← next[x]
if next[x] ≠ NIL
    then prev[next[x]] ← prev[x]
```

Час роботи O(1), але при видаленні за ключем спочатку треба викликати LIST_SEARCH (найгірший час $\Theta(n)$).

Елементарні структури даних. Зв’язані списки

- *Обмежувач* (sentinel) – фіктивний об’єкт, що спрощує обробку граничних умов.
- Введемо об’єкт *nil[L*], що представлятиме значення NIL, і помістимо його між головою і хвостом. Отримали *циклічний двозв’язний список з обмежувачем*.
- Тоді на голову списку вказуватиме *next*[*nil*[*L*]], а на хвіст – *prev*[*nil*[*L*]].
- Порожній список міститиме тільки обмежувач.
- Приклад використання: реалізація Introsort зі стандартної бібліотеки C++.



Елементарні структури даних. Зв’язані списки

- Модифікуємо наведені вище процедури:

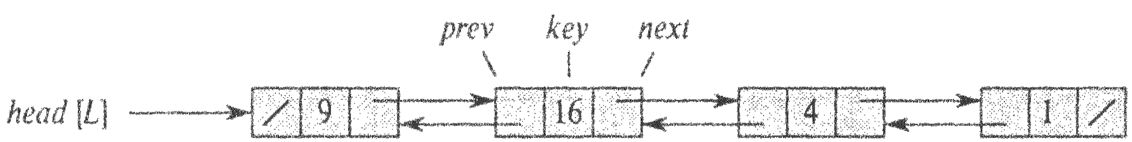
```
LIST_DELETE′(L, x)
  next[prev[x]] ← next[x]
  prev[next[x]] ← prev[x]

LIST_SEARCH′(L, k)
  x ← next[nil[L]]
  while x ≠ nil[L] и key[x] ≠ k
        do x ← next[x]
  return x

LIST_INSERT′(L, x)
  next[x] ← next[nil[L]]
  prev[next[nil[L]]] ← x
  next[nil[L]] ← x
  prev[x] ← nil[L]
```

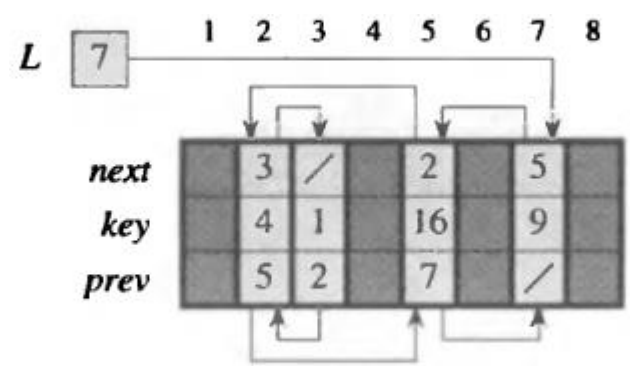
- Обмежувачі слід використовувати обдумано. Так, при обробці великої кількості таких малих списків може виникнути перевитрата пам’яті.

Реалізація вказівників і об’єктів

<u>Представлення об’єктів кількома масивами</u>

- Кожному полю відповідає свій масив.
- Для представлення двозв’язного списку потрібно три масиви: *key* для збереження ключів, *next* та *prev* для вказівників на наступний та попередній елементи. В ролі вказівників – індекси елементів, NIL позначають числом, яке не може бути індексом масиву, змінна L зберігає індекс голови.
- Таким чином, елементи *key*[x], *next*[x] та *prev*[x] представляють один об’єкт списку.





Реалізація вказівників і об’єктів

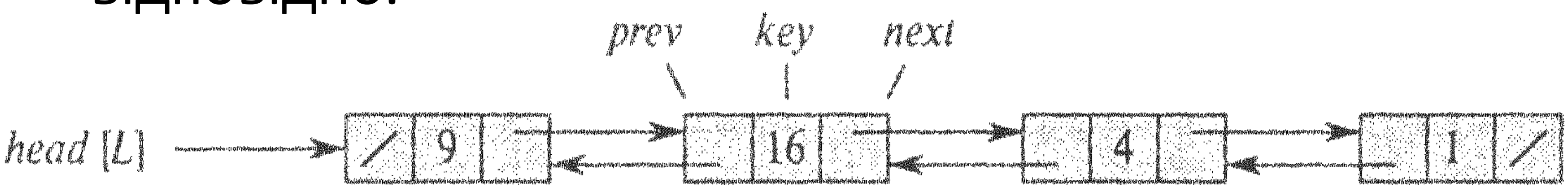
<u>Представлення об’єктів одним масивом</u>

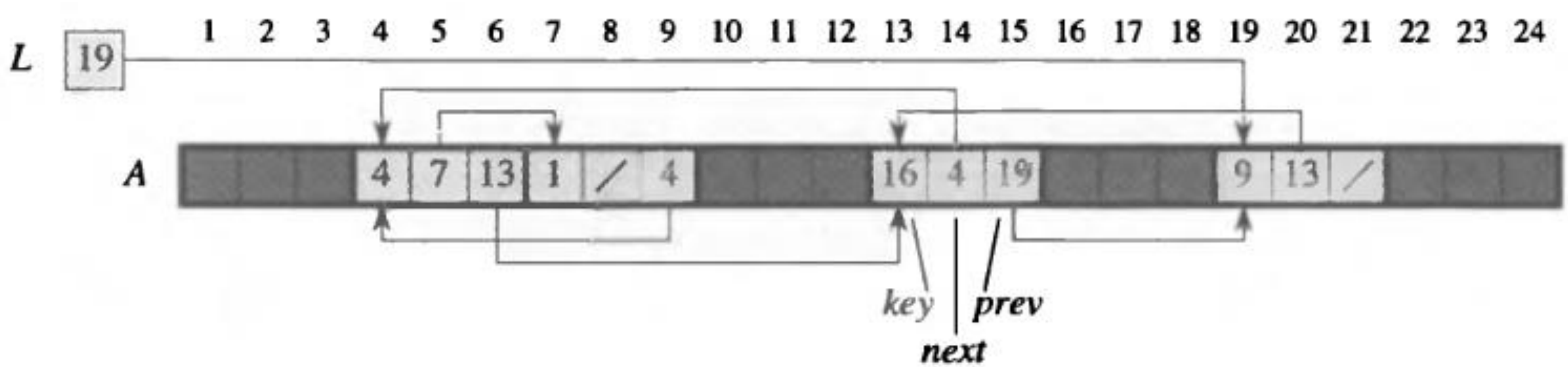
- Об’єкт займає неперервну ділянку пам’яті.
- Вказівником є індекс першої комірки пам’яті, де починається об’єкт.
- Індекс інших полів визначається за величиною зміщення.
- Таке представлення більш гнучке, оскільки дозволяє в одному масиві зберігати об’єкти різної довжини. Але водночас управління такими масивами є більш складною задачею.

Реалізація вказівників і об’єктів

<u>Представлення об’єктів одним масивом</u>

- Об’єкт займає підмасив A[j..k].
- Вказівником на об’єкт є індекс j, кожне поле відповідає зміщенню величиною від 0 до (k–j).
- Зміщення для полів *key*, *next* та *prev* – значення 0, 1 і 2 відповідно.





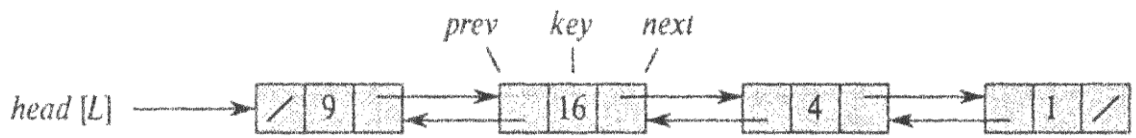
Управління пам’яттю

Розглянемо задачу виділення і звільнення пам’яті для однорідних об’єктів на прикладі двозв’язного списку, реалізованого декількома масивами.

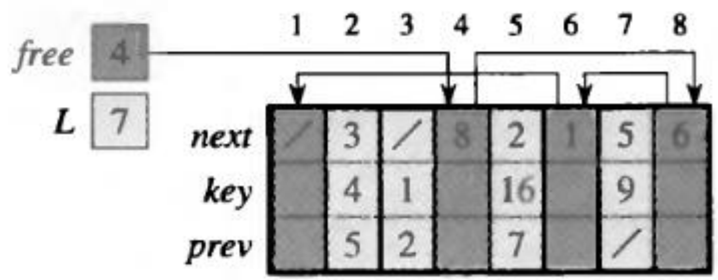
- Використовуються масиви довжиною m. В певний момент часу динамічна множина містить $n \le m$ елементів. Тому $(m-n)$ елементів вільні, їх можна використати в майбутньому для представлення нових елементів множини.
- Вільні елементи зберігаються в *списку вільних позицій* – однозв’язному списку, що використовує масив *next*. Індекс голови зберігає глобальна змінна *free*.
- Список вільних позицій – це стек: під кожен новий об’єкт виділяється остання звільнена пам’ять.

Управління пам’яттю

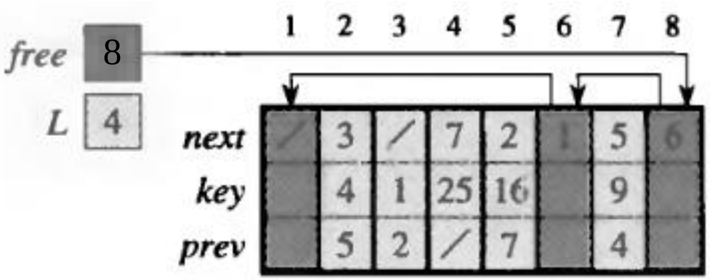
```
ALLOCATE_OBJECT()
if free = NIL
    then error "Нехватка памяти"
    else x ← free
         free ← next[x]
         return x
```



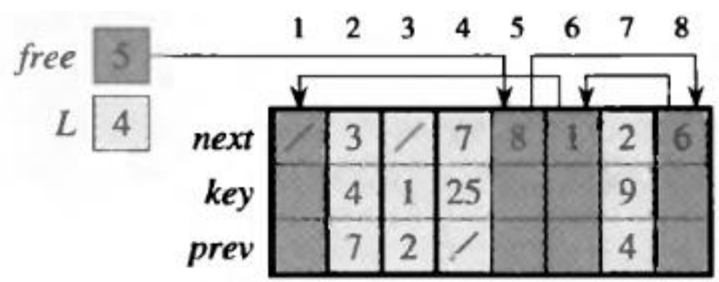

```
FREE_OBJECT(x)
next[x] ← free
free ← x
```



(а)



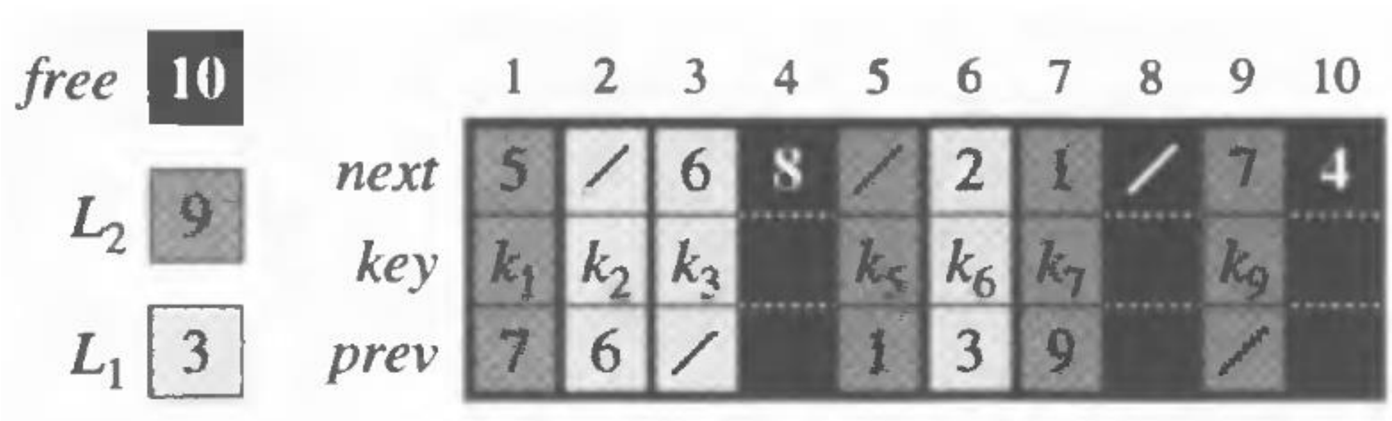
(б)



(в)

Управління пам’яттю

- Часто один список вільних позицій обслуговує декілька зв’язаних списків, що зберігаються в одному масиві:



- Час виконання обох процедур O(1).
- Їх можна модифікувати для роботи з довільним набором однорідних об’єктів, щоб тільки одне з полів працювало в якості поля *next* списку вільних позицій.

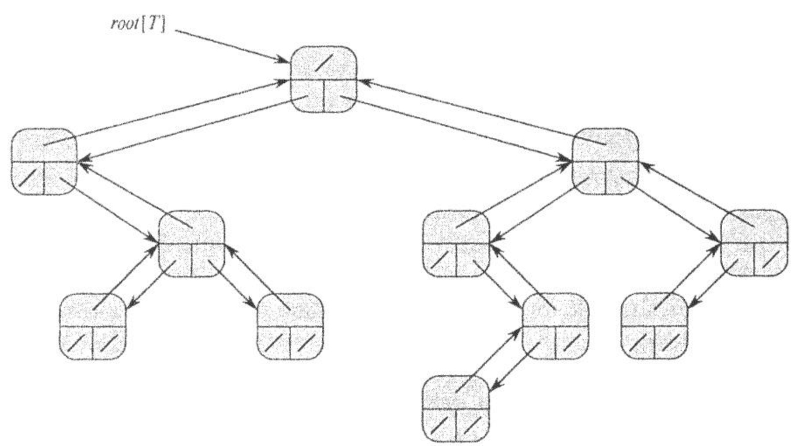
Представлення дерев з коренем

- В найзагальнішому випадку (*вільним*) *деревом* називають зв'язний неорієнтований ациклічний граф.
- Але для широкого практичного застосування така структура має бути більш структурованою: вводяться направленість та (можливо) впорядкування.
- *Направленість*: наявність кореня.
- *Впорядкування*: піддерева-сини однієї вершини мають певний порядок.
- Тому найчастіше коли йдеться про дерево, мається на увазі саме *дерево з коренем* і скоріш за все упорядковане.
- Таким чином, бінарне дерево з єдиним лівим піддеревом і бінарне дерево з єдиним правим піддеревом – різні.

Представлення дерев з коренем

<u>Бінарні дерева:</u>

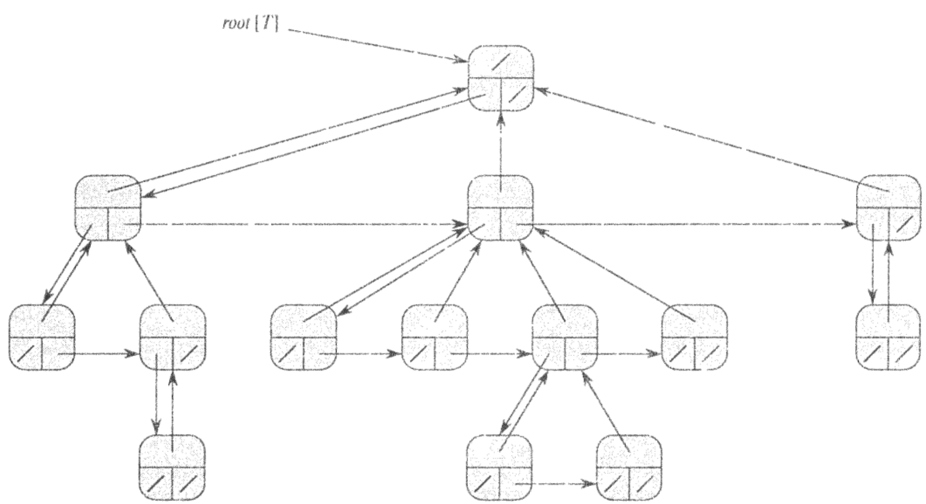
- поле-ключ *key*;
- вказівники на батька, лівого та правого синів *p*, *left*, *right*;
- x – корінь: *p*[x]=NIL;
- x – лист: *left*[x]=*right*[x]=NIL;
- *root*[T] – вказівник на корінь дерева T;
- *root*[T] = NIL – порожнє дерево.



Представлення дерев з коренем

<u>Довільна кількість вузлів (через бінарне дерево):</u>

- представлення з лівим дочірнім і правим сестринським вузлами;
- для дерева з n вузлів потрібно O(n) пам'яті;
- поле *left_child*[x] зберігає вказівник на найлівішого сина x; *left_child*[x]=NIL – лист;
- поле *right_sibling*[x] зберігає вказівник на правого брата; *right_sibling*[x]=NIL – x є найправішим сином.



Хешування і хеш-таблиці

- Ефективна структура даних для реалізації словників (асоціативних масивів).
- Узагальнення звичайного масиву.
- В середньому всі базові операції словника вимагають час O(1).

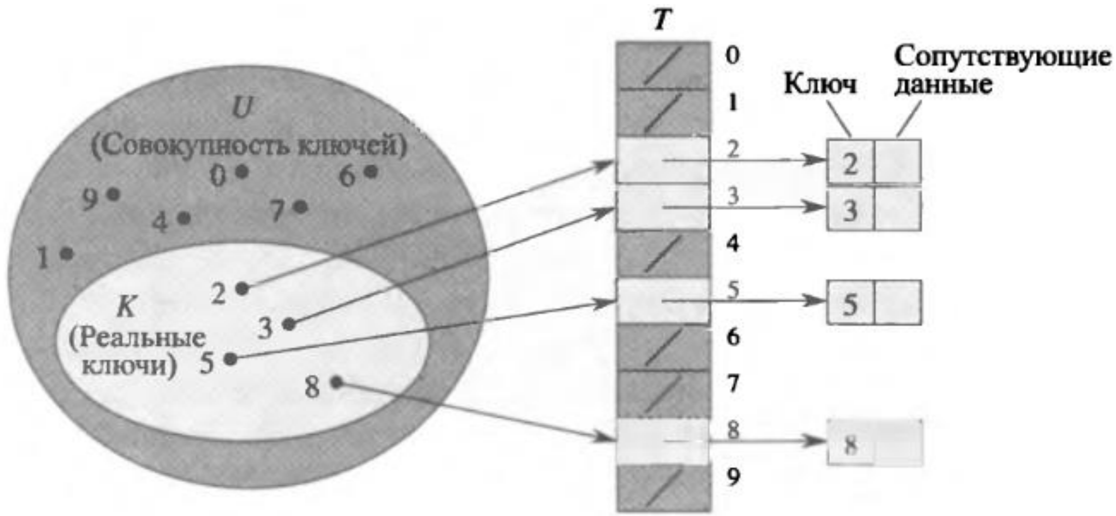
Хешування і хеш-таблиці

- Ефективна структура даних для реалізації словників (асоціативних масивів).
- Узагальнення звичайного масиву.
- В середньому всі базові операції словника вимагають час O(1):
  – пошук,
  – вставка,
  – видалення.

Таблиці з прямою адресацією

- Кожен елемент множини має ключ з множини U={0, 1, …, m–1}, де m невелике.
- Всі елементи мають різні ключі.
- Використовується масив T[0..m–1], кожна позиція (комірка) якого відповідає ключу з простору ключів U.
- Комірка k вказує на елемент множини з ключем k.
- T[k]=NIL: в множині елемент з ключем k відсутній.
- Іноді елементи зберігаються прямо в таблиці, що економить пам’ять. При цьому потрібний інший механізм позначення порожніх комірок.
- В середньому всі базові операції словника вимагають час O(1).

Таблиці з прямою адресацією



$\text{DIRECT\_ADDRESS\_SEARCH}(T, k)$
  return $T[k]$

$\text{DIRECT\_ADDRESS\_INSERT}(T, x)$
  $T[key[x]] \leftarrow x$

$\text{DIRECT\_ADDRESS\_DELETE}(T, x)$
  $T[key[x]] \leftarrow \text{NIL}$

- Кожна з операцій виконується за час O(1).

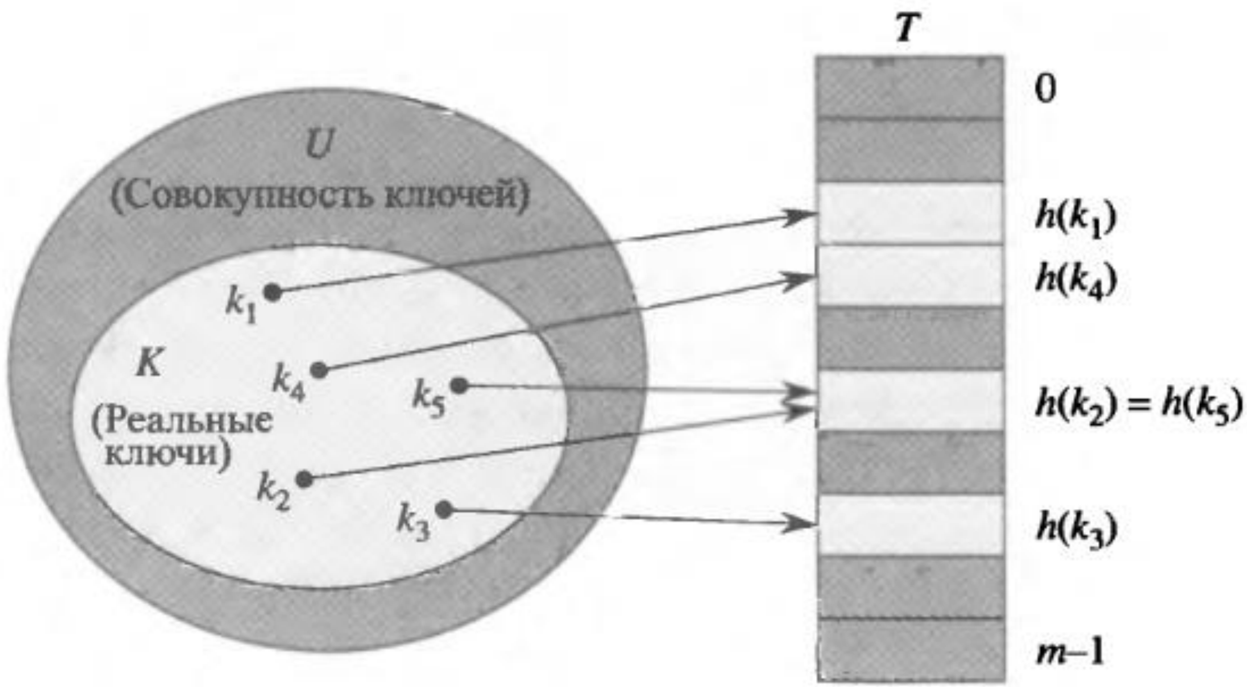
Хеш-таблиці

- Кількість реально збережених ключів може бути мала відносно простору можливих ключів U або кількість елементів в U завелика.
- При хешуванні елемент з ключем k зберігатиметься в комірці h(k): елемент з ключем k хешується в комірку h(k). Величина h(k) – *хеш-значення* ключа k.
- *Хеш-функція* h відображає простір ключів U на комірки хеш-таблиці T[0..m–1]:

$$h : U \to \{0, 1, \ldots, m-1\} .$$

- Мета хеш-функції – зменшити робочий діапазон масиву з |U| до m значень.
- Хеш-функція детермінована: для однакових k має давати те саме хеш-значення h(k).

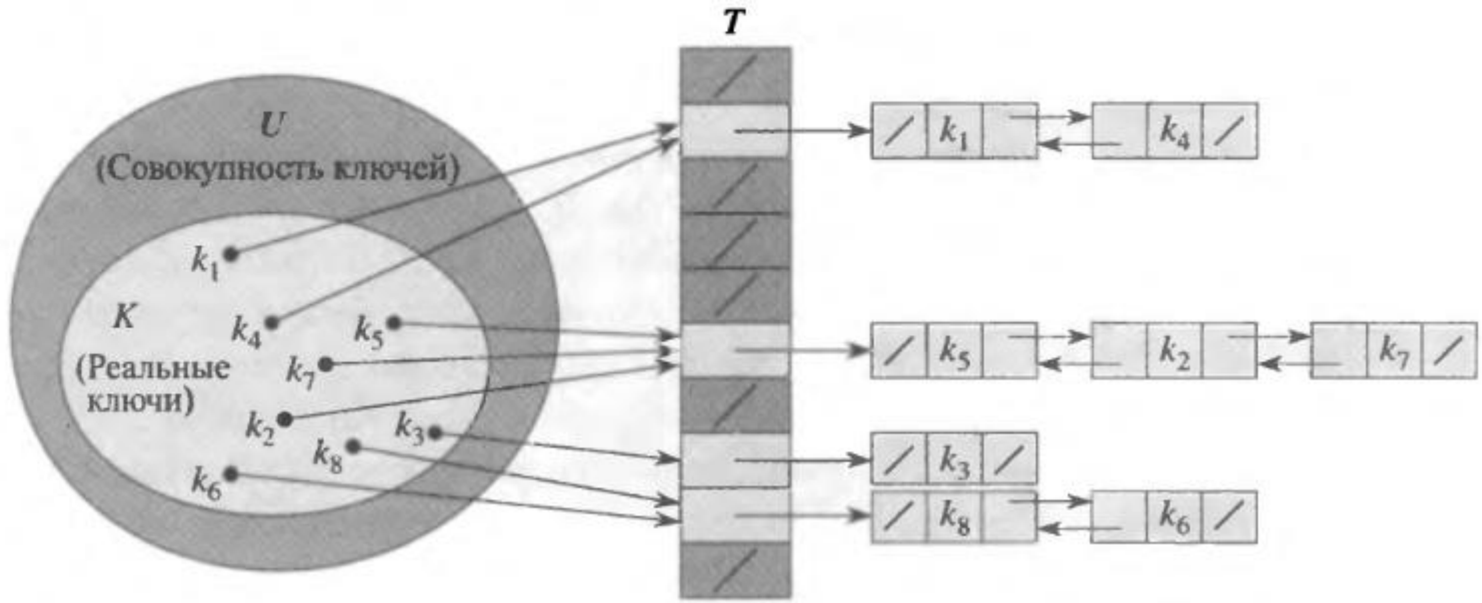
Хеш-таблиці



- *Колізія*: коли два ключі потрапили в одну комірку.
- Колізії будуть існувати «за побудовою», однак хороша хеш-функція може мінімізувати їх кількість.

Розв’язання колізій за допомогою ланцюжків

- Елементи з однаковими хеш-значеннями організовуються в список:



- CHAINED_HASH_INSERT(T,x): вставити x в голову списку T[h(*key*[x])].
- CHAINED_SEARCH(T,k): пошук елемента з ключем k в списку T[h(k)].
- CHAINED_HASH_DELETE(T,x): видалення x зі списку T[h(*key*[x])].

Хеш-функції

- Хороша хеш-функція має рівноймовірно поміщати ключ в одну з m комірок незалежно від хешування інших ключів.
- На практиці часто використовуються різні евристики.
- При побудові хеш-функцій дуже допомагає інформація про розподіл ключів.
- Функція має бути підібрана так, щоб не корелювала з можливими закономірностями вхідних даних.
- Іноді вимагається, щоб «близькі» ключі давали «далекі» хеш-значення.
- Для більшості хеш-функцій простір ключів представляється множиною натуральних чисел (або може бути певним чином так проінтерпретований).

Метод поділу

- Хеш-функція має вигляд

$$h(k) = k \bmod m.$$

- Деякі значення m будуть невдалими.
- Зокрема, значеннями m не беруть степені двійки.
- Хороші результати дають значення m, що є простими числами, далекими від степені 2.

Метод множення

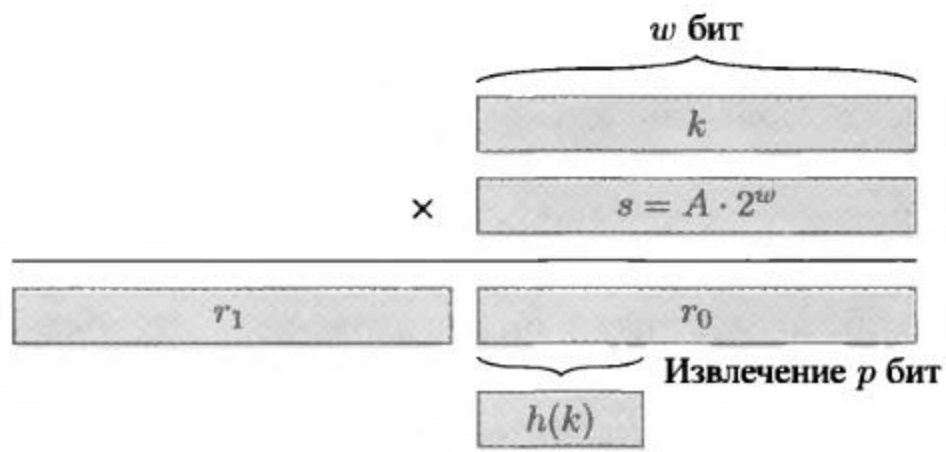
1. Ключ k множиться на деяку константу 0<A<1 і береться дробова частина.
2. Отримане значення домножується на m і відкидається дріб:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- Тепер значення m стає некритичним. Для зручності реалізації за m можна взяти степінь 2.

Метод множення

- Нехай розмір машинного слова w бітів, і k вміщується в одне слово. Нехай A має вигляд $s/2^w$, де $0<s<2^w$ та s ціле. Множимо k на w-бітове ціле $s=A\cdot 2^w$. Результат – 2w-бітове число $r_1 2^w+r_0$, де r_1 – старше слово, а r_0 – молодше. Старші p бітів числа r_0 є шуканим p-бітовим хеш-значенням.



- Д.Кнут запропонував $A \approx \left(\sqrt{5}-1\right)/2 \approx 0.6180339887\ldots$

Універсальне хешування

- Будь-яка фіксована хеш-функція має ризик стати вразливою у випадку підбору таких значень, які будуть хешуватися в одну комірку, призводячи до лінійного часу вибірки.
- Вихід – довільний вибір хеш-функції з деякої множини.
- Нехай H – скінченна множина хеш-функцій з простору ключів U в діапазон $\{0,1,\ldots,m-1\}$. Така множина *універсальна*, якщо для довільної пари хеш-ключів $k,l \in U$ кількість функцій $h \in H$, для яких $h(k)=h(l)$, не перевищує $|H|/m$.
- Тобто ймовірність колізії не перевищує ймовірності співпадіння двох серед m різних елементів.
- Універсальні хеш-функції мають хорошу ефективність.

Універсальне хешування

Побудуємо універсальну множину хеш-функцій.

- Візьмемо просте p, щоб всі можливі ключі входили до діапазону 0..(p–1).
- Нехай $Zp=\{0,1,\ldots,p-1\}$, $Zp^*=\{1,2,\ldots,p-1\}$.
- $m<p$ – кількість комірок в хеш-таблиці.
- Визначимо хеш-функцію $h_{a,b}$ для всіх $a\in Zp^*$, $b\in Zp$ так

$$h_{a,b}(k)=((ak+b) \bmod p) \bmod m.$$

Наприклад, при $p = 17$, $m = 6$ маємо $h_{3,4}(8) = 5$.

- Сімейство таких функцій – множина

$$H_{p,m}=\{h_{a,b} : a\in Zp^*, b\in Zp\}$$

- Множина визначених таким способом хеш-функцій універсальна; вона містить $p(p-1)$ функцій.

Відкрита адресація

- Всі елементи зберігаються безпосередньо в хеш-таблиці: записи таблиці містять або елемент динамічної множини, або значення NIL.
- Шукаючи елемент, систематично перевіряються комірки, поки він не знайдеться або не впевнимося в його відсутності.
- Можлива ситуація заповнення таблиці і неможливості вставки нового елемента.
- Вказівники відсутні. Обчислюється послідовність комірок, які переглядаються.

Відкрита адресація

- При пошуку і вставці ключа послідовно перевіряються комірки хеш-таблиці, поки не знаходиться порожня комірка, куди вставляється новий елемент (або комірка з шуканим елементом).
- Послідовність перевірки залежить від *конкретного ключа*.
- Побудуємо розширену номером перевірки хеш-функцію:

$$h: U \times \{0,1,\ldots,m-1\} \to \{0,1,\ldots,m-1\}.$$

- Необхідно, щоб для кожного ключа послідовність перевірок $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ була перестановкою множини $\{0,1,\ldots,m-1\}$ – тобто щоб були проглянуті всі комірки.

Відкрита адресація

```
HASH_INSERT(T, k)
i ← 0
repeat  j ← h(k, i)
        if T[j] = NIL
          then T[j] ← k
               return j
          else  i ← i + 1
   until i = m
error "Хеш-таблица переполнена"
```

```
HASH_SEARCH(T, k)
i ← 0
repeat  j ← h(k, i)
        if T[j] = k
          then return j
        i ← i + 1
   until T[j] = NIL или i = m
return NIL
```

- Процедура видалення складна – не можна просто позначити комірку як NIL.

Відкрита адресація

- Множина $\{0,1,\ldots,m-1\}$ має $m!$ перестановок, але існуючі методи дають не більше m^2 варіантів.

Лінійне дослідження

Візьмемо звичайну хеш-функцію $h': U \to \{0,1,\ldots,m-1\}$.

Назвемо її *допоміжною хеш-функцією*. Для методу задамо хеш-функцію так:

$$h(k,i) = (h'(k)+i) \bmod m,$$

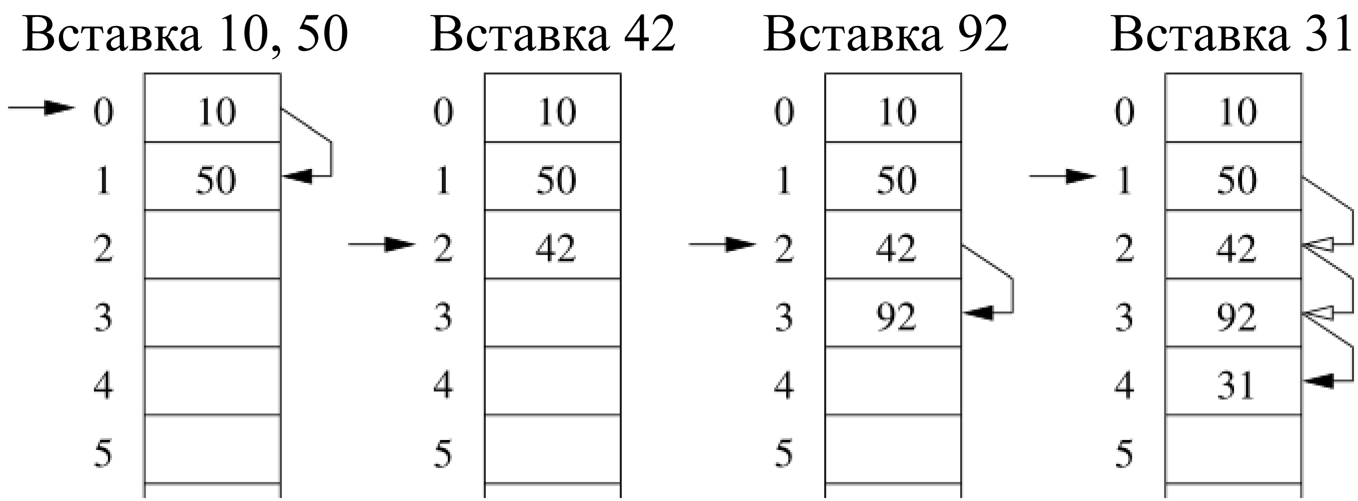
де i приймає значення від 0 до $m-1$.

Проблема первинної кластеризації: утворюються довгі ланцюжки зайнятих комірок.

Метод дає m перестановок.

Відкрита адресація

Приклад виникнення кластеризації при лінійному дослідженні (показані перші 6 комірок).



Послідовно вставляються ключі 10, 50, 42, 92, 31.

Допоміжна хеш-функція має вигляд h′(k) = k mod 10.

Відкрита адресація

Квадратичне дослідження

- Хеш-функція задається так:

$$h(k,i) = (h'(k)+c_1 i+c_2 i^2) \bmod m,$$

де h' – допоміжна хеш-функція, $c_1,c_2 \neq 0$ – допоміжні константи, а *i* приймає значення від 0 до $m-1$.

- Необхідно вибирати значення c_1,c_2 та m.
- Проблема вторинної кластеризації: якщо два ключі мають одну й ту саму початкову позицію, то послідовності дослідження також співпадатимуть.
- Метод дає m перестановок.

Відкрита адресація

Подвійне хешування

- Один з найкращих способів відкритої адресації.
- Метод дає m^2 перестановок.
- Хеш-функція задається так:

$$h(k,i) = (h_1(k)+ih_2(k)) \bmod m,$$

де h_1,h_2 – допоміжні хеш-функції.

- Таким чином, початкова позиція і крок обчислюються окремо.
- Щоб обійти всю таблицю необхідно, щоб значення $h_2(k)$ було взаємно простим з розміром хеш-таблиці m (m – степінь 2 і тільки непарні значення $h_2(k)$ або m просте, а $h_2(k)$ повертає значення, менші за m).

Відкрита адресація

<u>Подвійне хешування</u>

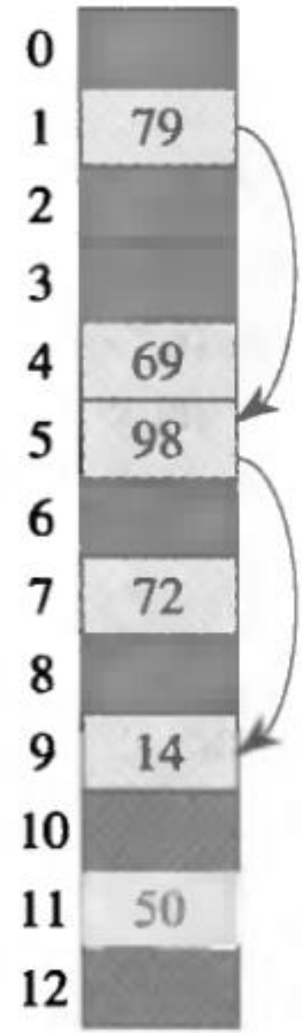
Приклад вибору хеш-функцій:

$h_1(k)=k \bmod m$

$h_2(k)=1+(k \bmod m')$,

де m просте, а m′ трохи менше за m

(як варіант, $m'=m-1$)



Приклад вставки ключа 14 в таблицю

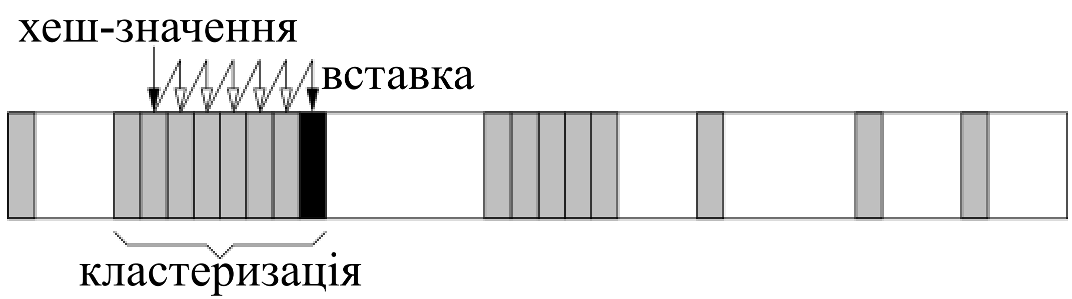
$h_1(k)=k \bmod 13$,

$h_2(k)=1+(k \bmod 11)$.

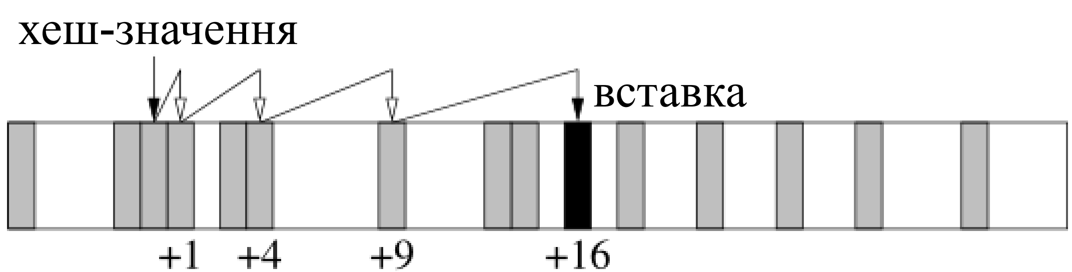
Відкрита адресація

Порівняння різних стратегій при відкритій адресації

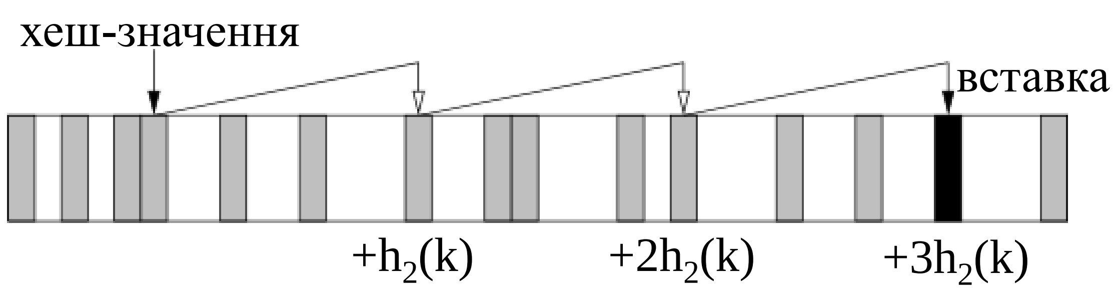
Лінійне дослідження



Квадратичне дослідження



Подвійне хешування



Ідеальне хешування

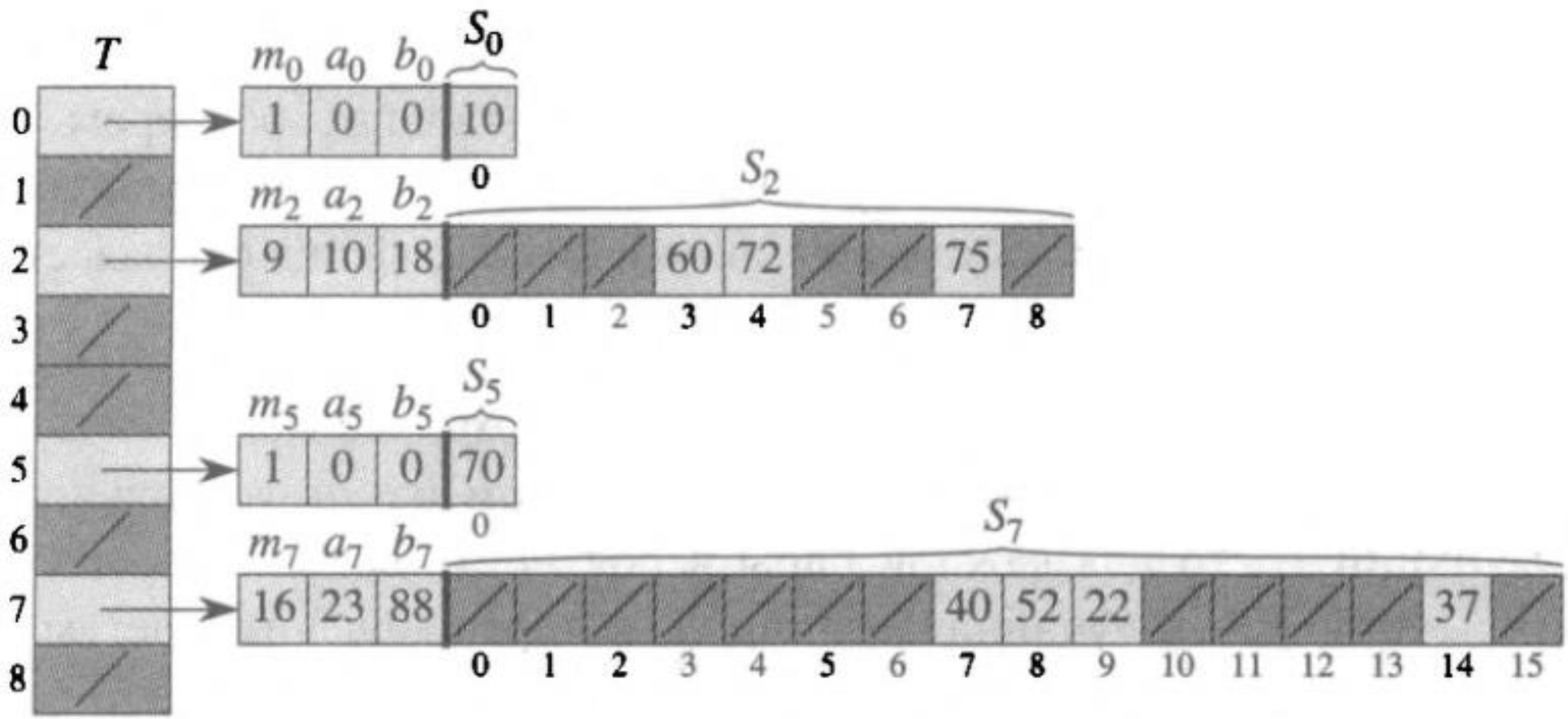
- O(1) звертань до пам’яті в найгіршому випадку.
- Множина ключів статична – не змінюється після збереження в таблицю.
- Перший рівень хешування: n ключів хешуються в m комірок за допомогою універсальної хеш-функції h.
- Другий рівень хешування: для кожної комірки своя вторинна хеш-таблиця зі своєю універсальною хеш-функцією, вибраною так, щоб уникнути колізій; її розмір – квадрат кількості ключів, захешованих в комірку.
- Очікувана загальна пам’ять під таку структуру O(n).

Ідеальне хешування

Приклад використання ідеального хешування.

Множина ключів K={10,22,37,40,52,60,70,72,75}

h(k)=((ak+b) mod p) mod m, де a=3, b=42, p=101, m=9



Запитання і завдання

- Покажіть, як реалізувати (1) чергу за допомогою двох стеків та (2) стек за допомогою двох черг. Проаналізуйте час роботи операцій, які виконуються з елементами.
- Часто (наприклад, при сторінковій організації віртуальної пам'яті) всі елементи списку необхідно розміщувати компактно, в неперервній ділянці пам'яті. Розробіть таку реалізацію процедур ALLOCATE_OBJECT та FREE_OBJECT, щоб елементи списка займали позиції 1..m, де m – кількість елементів у списку. Скористайтесь реалізацією стека на масиві.
- Розробіть процедуру, що за час O(n) виводитиме ключі всіх n вузлів довільного дерева з коренем, яке реалізоване в представленні з лівим дочірнім та правим сестринським елементами.
- Бітовий вектор є масивом бітів (нулів і одиниць). Бітовий вектор довжиною m займає істотно менше місця, ніж масив з m вказівників. Яким чином можна використовувати бітовий вектор для представлення динамічної множини різних елементів без супутніх даних? Словникові операції повинні виконуватися за час O(1).

Запитання і завдання

- Піраміда зі злиттям (mergable heap) підтримує операції Make_Heap (створення порожньої піраміди), Insert, Minimum, Extract_Minimum та Union. Реалізуйте піраміди зі злиттям для кожного з наступних випадків: (1) списки відсортовані; (2) списки не відсортовані; (3) списки не відсортовані і динамічні множини, що об’єднуються, не перетинаються. Намагайтеся, щоб кожна з операцій виконувалася максимально ефективно. Проаналізуйте час роботи кожної операції відносно розміру вхідної динамічної множини.
- Запропонуйте спосіб реалізації таблиці з прямою адресацією, в якій ключі елементів, що зберігаються, можуть збігатися, а самі елементи – мати супутні дані. Всі словникові операції мають виконуватися за час O(1). (Пам’ятайте, що аргументом процедури видалення є покажчик об’єкта).

Запитання і завдання

- Нехай ми хочемо реалізувати словник з використанням прямої адресації дуже великого масиву. Спочатку масиві може містити "сміття", але ініціалізація всього масиву нераціональна в силу його розміру. Розробіть схему реалізації такого словника. Кожен об'єкт, що зберігається, має використовувати O(1) пам'яті; словникові операції ті ініціалізація також мають виконуватися за час O(1). (Для визначення, чи є даний запис у великому масиві коректним чи ні, скористайтеся додатковим стеком, розмір якого дорівнює кількості збережених у словнику ключів.)
- Розглянемо варіант методу ділення при побудові хеш-функцій, при якому h(k)=k mod m, де $m=2^p-1$, а k – символьний рядок, що інтерпретується як ціле число в системі числення з основою 2^p. Покажіть, що якщо рядок x може бути отриманий з рядка y перестановкою символів то хеш-значення цих рядків однакові. Наведіть приклад, коли така властивість хеш-функції виявиться вкрай небажаною.