Программирование на языке высокого уровня

Модуль 5. Сложные структуры данных

Цой Ю.Р.
Кафедра вычислительной техники
Томский политехнический университет

Содержание

1. Стеки и очереди

Динамические множества

Множество – это фундаментальное понятие, как в математике, так и в теории вычислительных машин.

Множества, которые обрабатываются в ходе выполнения алгоритмов, могут с течением времени разрастаться, уменьшаться или подвергаться другим изменениям, будем называть *динамическими* (*dynamic*).

Элемент множества: <ключ, значение>

Динамические множества

Операции динамического множества

- *Запросы* (*queries*).
- *Модифицирующие операции* (*modifying operations*).

Типичные операции

Search (**S**, *k*)	Maximum (**S**)
Insert (**S**, *x*)	Successor (**S**, *x*)
Delete (**S**, x)	Predecessor (**S**, *x*)
Minimum (**S**)	

Стеки и очереди

Стеки и очереди – это динамические множества, элементы из которых удаляются с помощью предварительно определенной операции Delete.

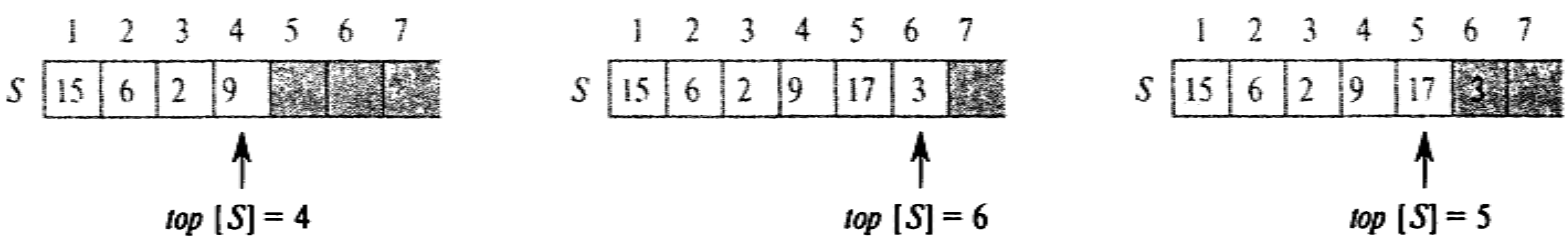
Стек (*stack*): стратегия «последним вошел – первым вышел» (last-in, first-out – LIFO)

Очередь (*queue*): стратегия «первым вошел – первым вышел» (first-in, first-out – FIFO)

Рассмотрим, как реализовать обе эти структуры данных с помощью обычного массива.

Стек

Вставка	Push
Удаление	Pop
top[**S**]	индекс последнего элемента
S[1]	элемент на дне стека
S[*top*[**S**]]	элемент на вершине стека
top[**S**] = 0	пустой (*empty*) стек



а) б) в)

Стек

- Если элемент снимается с пустого стека, говорят, что он *опустошается* (*underflow*).
- Если значение *top*[**S**] больше *n*, то стек *переполняется* (*overflow*)

```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else  return FALSE

PUSH(S, x)
top[S] ← top[S] + 1
S[top[S]] ← x

POP(S)
if STACK_EMPTY(S)
    then error "underflow"
    else  top[S] ← top[S] − 1
          return S[top[S] + 1]
```

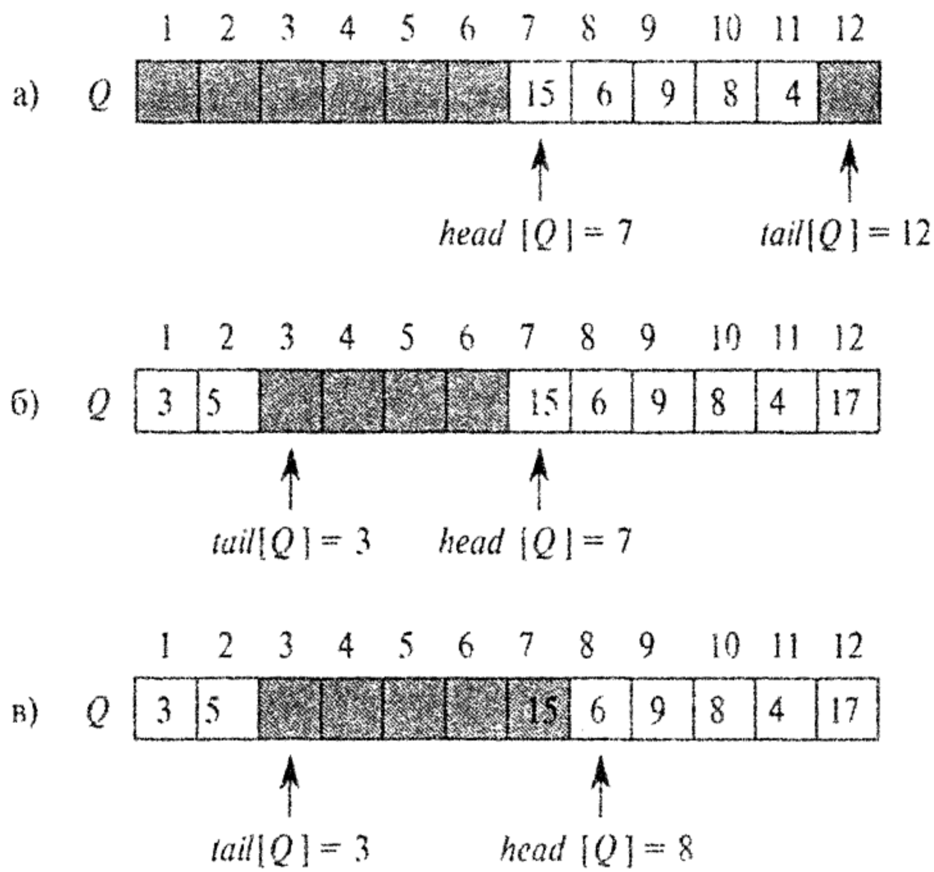
Любая из трех описанных операций со стеком выполняется в течение времени $O(1)$.

Упражнение:
Переписать, чтобы учесть переполнение стека.

Очередь

Вставка	Enqueue
Удаление	Dequeue
Голова , *head*[**Q**]	Начало очереди
Хвост, *tail*[**Q**]	Конец очереди
head[**Q**] = *tail*[**Q**] = 1	Начальное состояние
head[**Q**] = *tail*[**Q**]	пустая (*empty*) очередь
head[**Q**] = *tail*[**Q**]+1	очередь заполнена

Очередь



$\text{ENQUEUE}(Q, x)$

$Q[tail[Q]] \leftarrow x$
if $tail[Q] = length[Q]$
  then $tail[Q] \leftarrow 1$
  else $tail[Q] \leftarrow tail[Q] + 1$

$\text{DEQUEUE}(Q)$

$x \leftarrow Q[head[Q]]$
if $head[Q] = length[Q]$
  then $head[Q] \leftarrow 1$
  else $head[Q] \leftarrow head[Q] + 1$
return x

Упражнение:
Переписать, чтобы учесть опустошение и переполнение очереди.

Связанные списки

Связанный список (*linked list*) – это структура данных, в которой объекты расположены в линейном порядке.

Виды списков:

- *однократно связанный* (*однонаправленный, односвязный*) (*singly linked*)
- *дважды связанный (двусвязный)* (*doubly linked*)
- *отсортированный* (*sorted*)
- *неотсортированный* (*unsorted*)
- *кольцевой* (*circular list*)

Будем рассматриваться неотсортированные дважды связанные списки.

Поиск в связанном списке

$\text{LIST\_SEARCH}(L, k)$
1 $x \leftarrow head[L]$
2 **while** $x \neq \text{NIL}$ **и** $key[x] \neq k$
3 **do** $x \leftarrow next[x]$
4 **return** x

Поиск с помощью функции List_Search в списке, состоящем из *n* элементов, в наихудшем случае выполняется в течение времени $O(n)$, поскольку может понадобиться просмотреть весь список.

Вставка в связанный список

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

Время работы процедуры List_Insert равно $O(1)$.

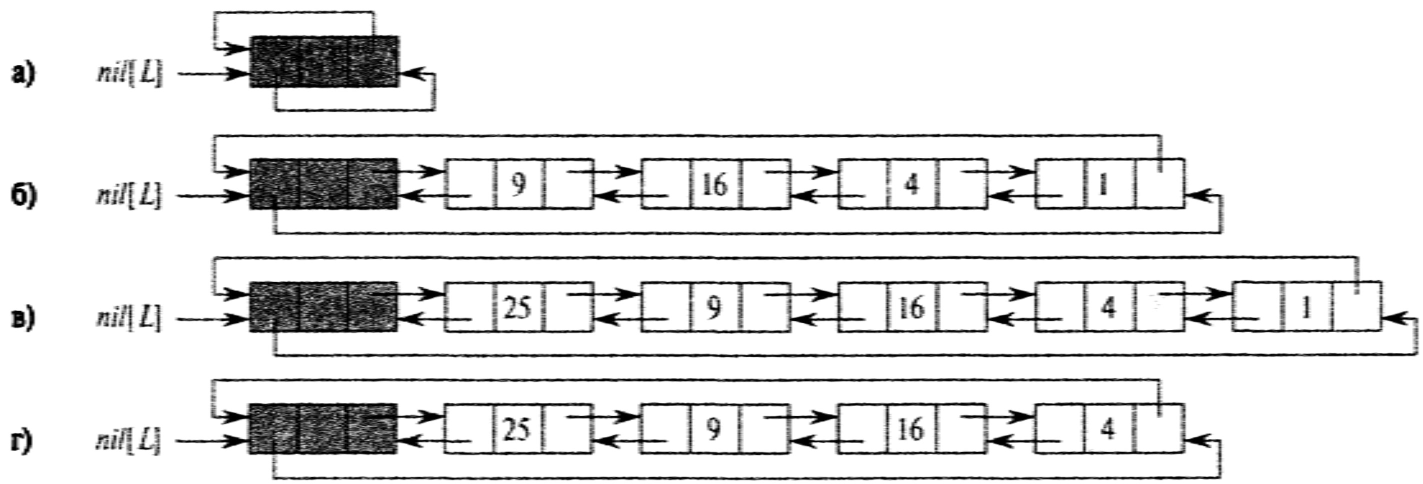
Удаление из связанного списка

```
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]
```

Время работы процедуры List_Delete равно *O*(1), но если нужно удалить элемент с заданным ключом, то в наихудшем случае понадобится время *O*(*n*), поскольку сначала необходимо вызвать процедуру List_Search.

Ограничители

Ограничитель (*sentinel*) – это фиктивный объект, упрощающий учет граничных условий.



Наличие ограничителя преобразует обычный дважды связанный список в *циклический дважды связанный список с ограничителем* (*circular, doubly linked list with a sentinel*).

Ограничители

LIST_INSERT′(L, x)

1 $next[x] \leftarrow next[nil[L]]$
2 $prev[next[nil[L]]] \leftarrow x$
3 $next[nil[L]] \leftarrow x$
4 $prev[x] \leftarrow nil[L]$

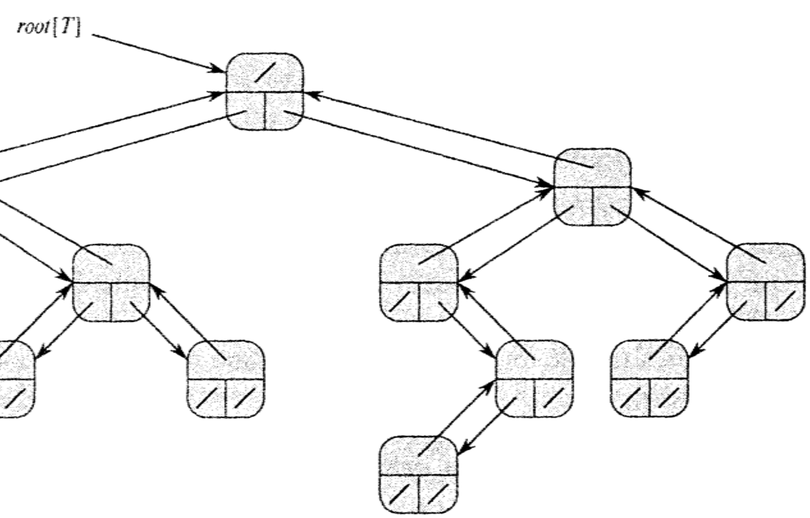
Особенности:

- Повышение скорости (уменьшение значений постоянных множителей в оценках трудоемкости операций).
- Повышается ясность и компактность кода.
- Увеличение объема занимаемой памяти.

2. Деревья

Бинарные деревья

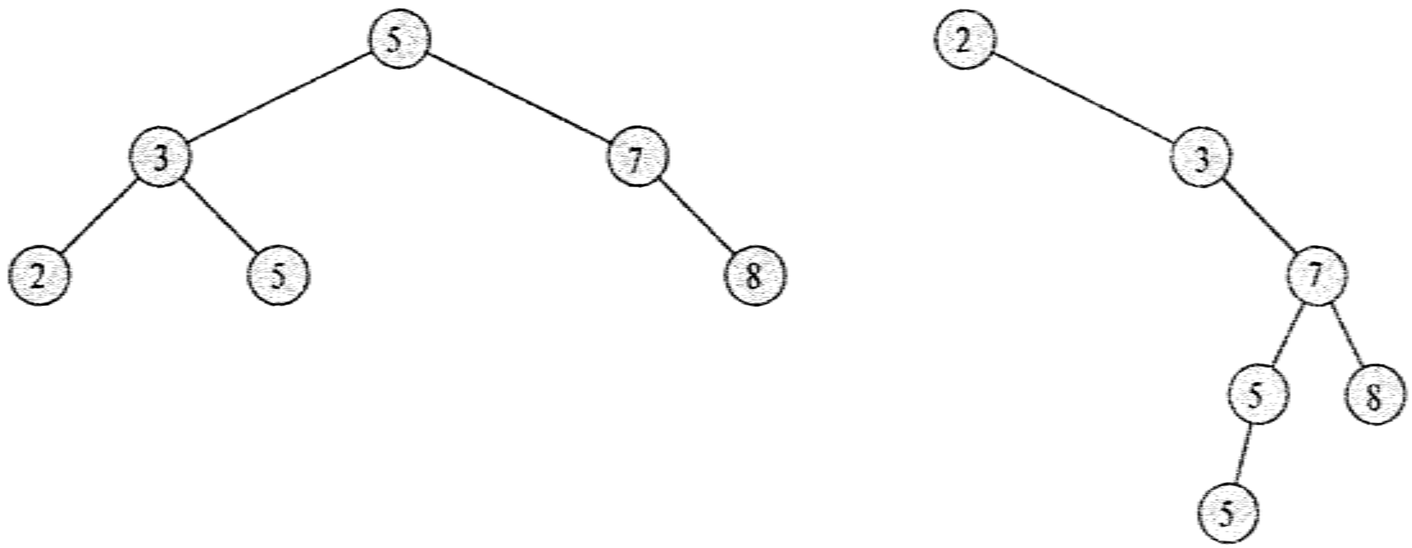
Узел дерева	Отдельный объект.
p	Указатель на родительский узел
left	Указатель на дочерний левый узел
right	Указатель на дочерний правый узел
root[*T*]	Корень дерева
p[*x*] = NULL	Корень дерева



Бинарные деревья поиска

- Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте.
- Математическое ожидание высоты построенного случайным образом бинарного дерева равно $O(\lg n)$, так что все основные операции над динамическим множеством в таком дереве выполняются в среднем за время $O(\lg n)$
- Случайность построения бинарного дерева поиска не всегда может быть гарантирована

Бинарные деревья поиска



Свойство бинарного дерева поиска:

Если *x* – узел бинарного дерева поиска, а узел *y* находится в левом поддереве *x*, то $key[y] \le key[x]$. Если узел *y* находится в правом поддереве *x*, то $key[x] \le key[y]$.

Бинарные деревья поиска

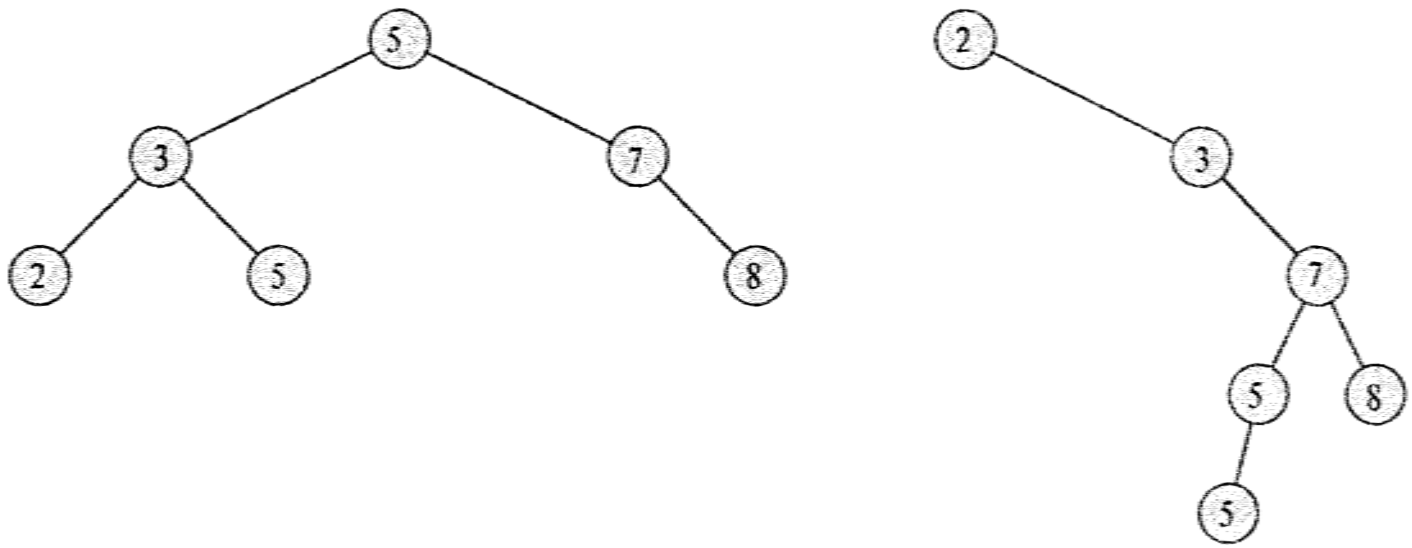
Способы обхода дерева:

- *центрированный (симметричный) обход* (*inorder tree walk*)
- *обход в прямом порядке* (*preorder tree walk*)
- *обход в обратном порядке* (*postorder tree walk*)

```
INORDER_TREE_WALK(x)
if x ≠ NIL
   then INORDER_TREE_WALK(left[x])
        print key[x]
        INORDER_TREE_WALK(right[x])
```

Упражнение: Реализовать прямой и обратный обходы дерева.

Бинарные деревья поиска



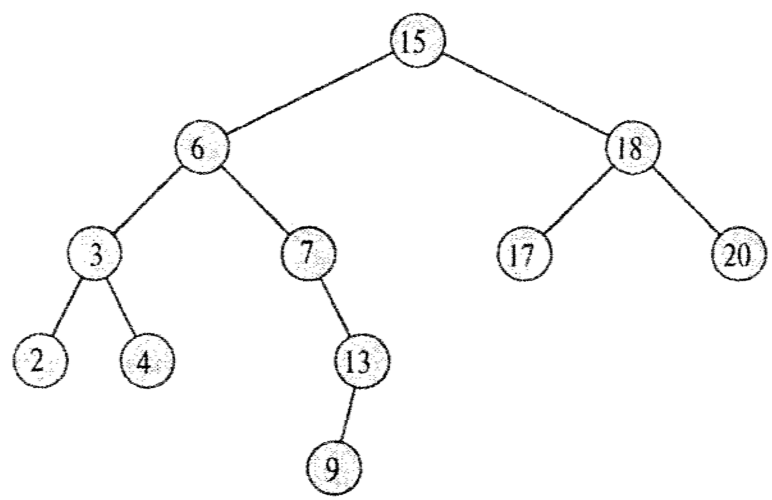
- Результат симметричного обхода:

  2, 3, 5, 5, 7, 8

Теорема. Если *x* – корень поддерева, в котором имеется *n* узлов, то процедура Inorder_Tree_Walk(*x*) выполняется за время $\Theta(n)$.

Поиск

```
TREE_SEARCH(x, k)
if x = NIL или k = key[x]
    then return x
if k < key[x]
    then return TREE_SEARCH(left[x], k)
    else return TREE_SEARCH(right[x], k)
```



Подпись узла с ключом 13: $15 \to 6 \to 7 \to 13$

Поиск

```
ITERATIVE_TREE_SEARCH(x, k)
while x ≠ NIL и k ≠ key[x]
      do if k < key[x]
            then x ← left[x]
            else  x ← right[x]
return x
```

Нерекурсивный вариант процедуры поиска

Поиск минимума и максимума

```
TREE_MINIMUM(x)
while left[x] ≠ NIL
      do x ← left[x]
return x
```

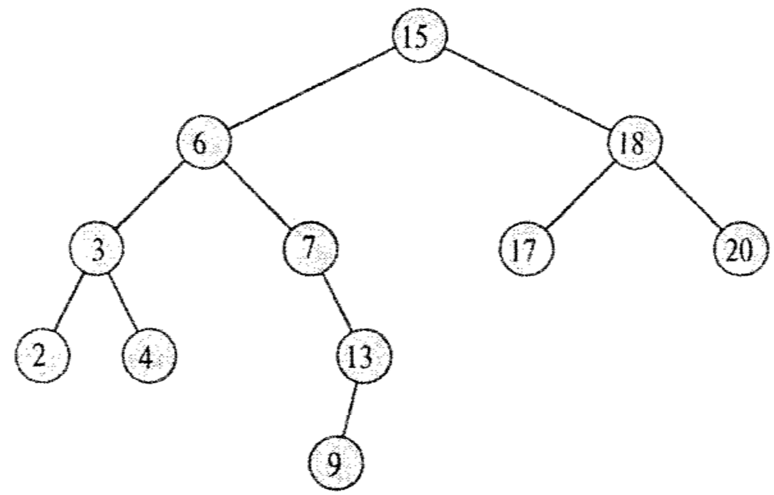
```
TREE_MAXIMUM(x)
while right[x] ≠ NIL
      do x ← right[x]
return x
```

Корректность процедур поиска гарантируется свойством бинарного дерева.

Время поиска равно *O*(*h*), где *h* – высота дерева

Предшествующий и последующий элементы

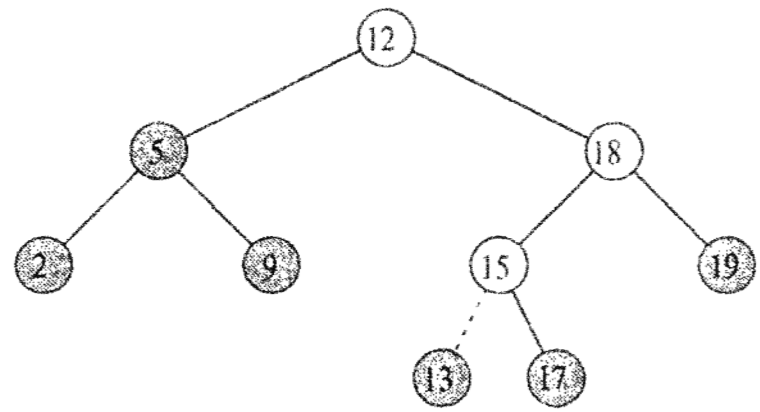
```
TREE_SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE_MINIMUM(right[x])
y ← p[x]
while y ≠ NIL и x = right[y]
      do x ← y
         y ← p[y]
return y
```



Теорема. Операции поиска, определения минимального и максимального элемента, а также предшествующего и последующего, в бинарном дереве поиска высоты *h* могут быть выполнены за время $O(h)$.

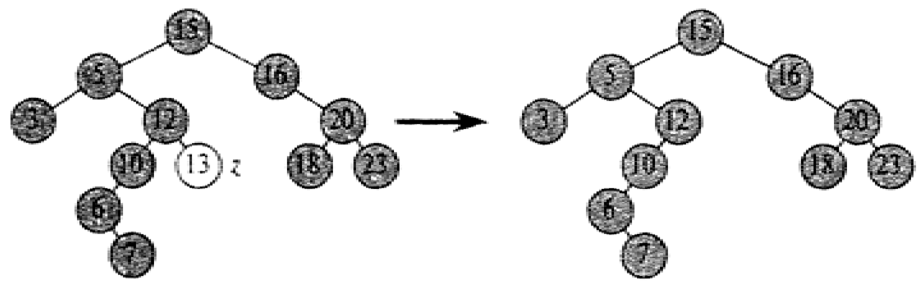
Вставка элемента

```
TREE_INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                 ▷ Дерево T — пустое
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```



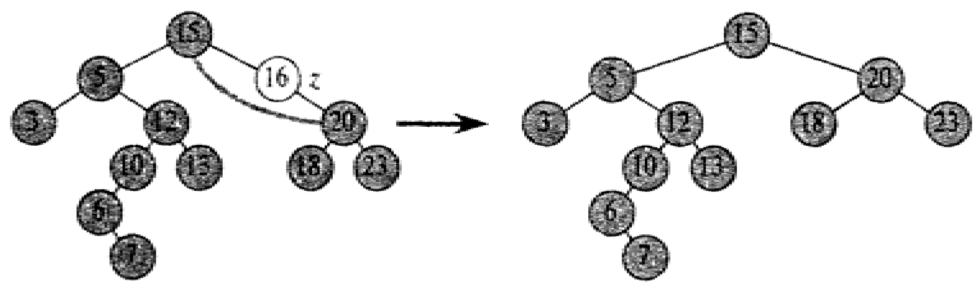
Удаление элемента

Нет дочерних узлов



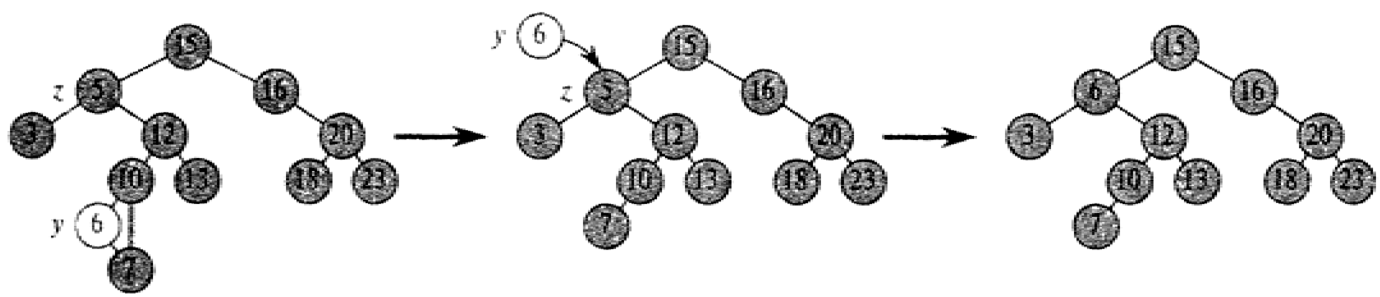
а)

Один дочерний узел



б)

Два дочерних узла



в)

Удаление элемента

```
TREE_DELETE(T, z)
if left[z] = NIL или right[z] = NIL
   then y ← z
   else y ← TREE_SUCCESSOR(z)
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
           Копирование сопутствующих данных в z
return y
```

Теорема. Операции вставки и удаления в бинарном дереве поиска высоты *h* могут быть выполнены за время *O*(*h*).

Корневые деревья с произвольным ветвлением

Схему представления бинарных деревьев можно обобщить для деревьев любого класса, в которых количество дочерних узлов не превышает некоторой константы k. При этом поля *right* и *left* заменяются полями $child_1$, $child_2$, …, $child_k$.

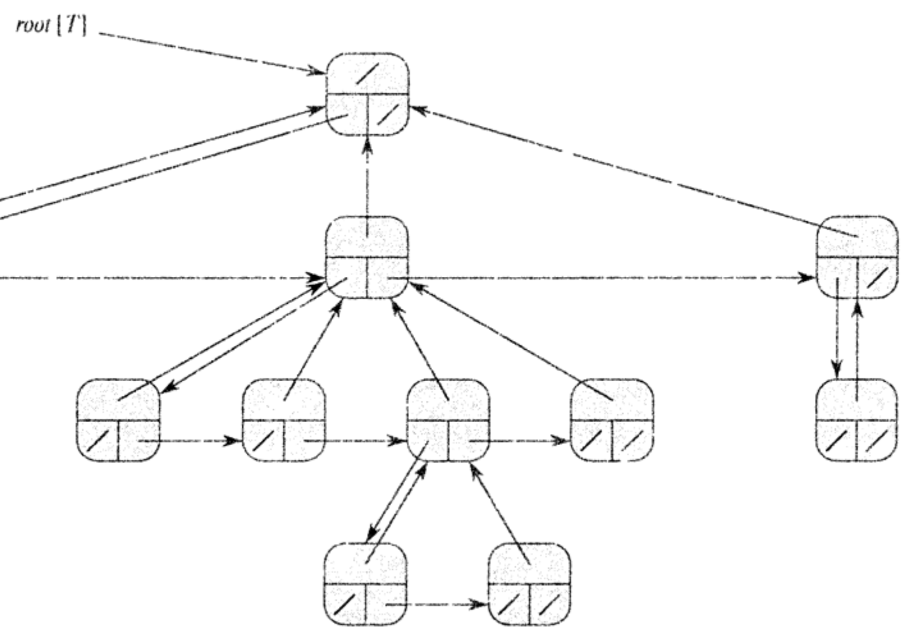
Недостатки:

- схема не работает, если количество дочерних элементов узла не ограничено;
- если количество дочерних элементов k ограничено большой константой, то значительный объем памяти расходуется напрасно.

Корневые деревья с произвольным ветвлением

Схема представления деревьев с произвольным количеством дочерних узлов с помощью бинарных деревьев.

Представление *с левым дочерним и правым сестринским узлами* (*left-child, right-sibling representation*)



Корневые деревья с произвольным ветвлением

Каждый узел *x* содержит всего два указателя:

- В поле *left_child*[*x*] хранится указатель на крайний левый дочерний узел узла *x*.
- В поле *right_sibling*[*x*] хранится указатель на узел, расположенный на одном уровне с узлом *x* справа от него.

Если узел *x* не имеет потомков, то *left_child*[*x*] = NULL, а если узел *x* – крайний правый дочерний элемент какого-то родительского элемента, то *right_sibling*[*x*] = NULL.

Существуют и другие способы.

3. Хеш-таблицы

Хеш-таблица (*hash table*) представляет собой эффективную структуру данных для реализации словарей. Является обобщением обычного массива.

Поиск элемента в наихудшем случае требует $O(n)$ операций. Однако в большинстве случаев среднее время поиска элемента в хеш-таблице составляет $O(1)$.

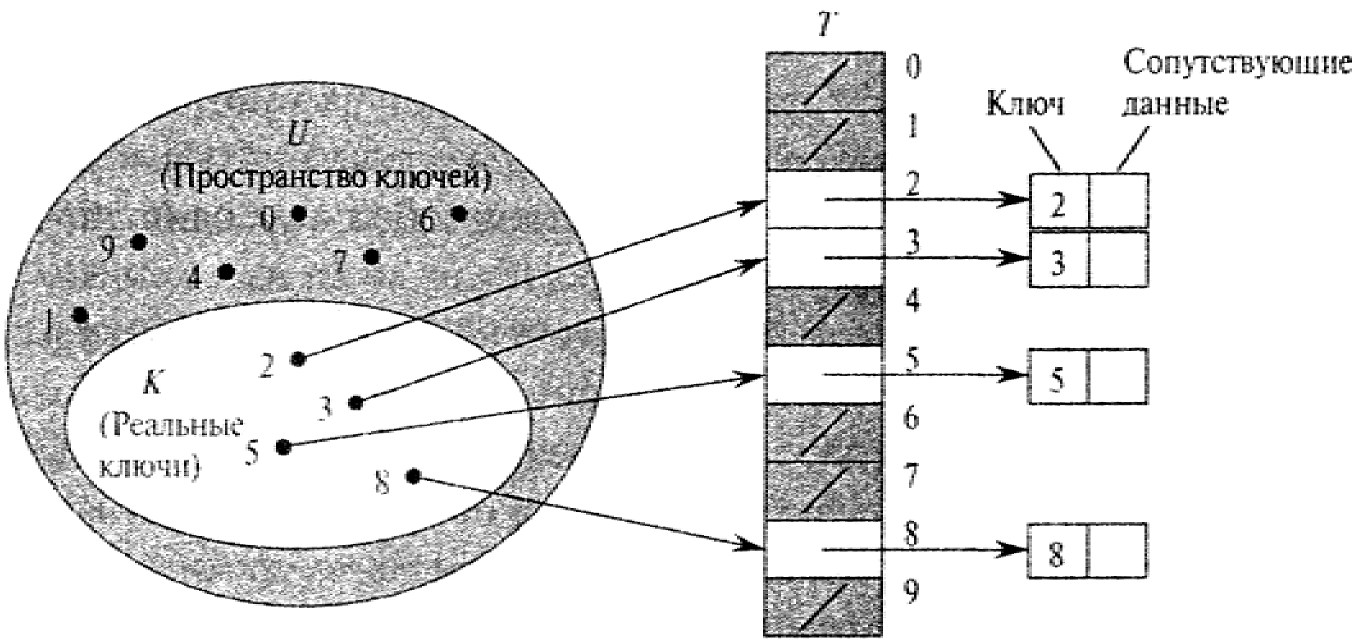
Таблицы с прямой адресацией

Представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей.

Предположим, что требуется динамическое множество, каждый элемент которого имеет ключ из множества **U** = {0,1,..., m – 1}, где m не слишком велико.

Используем массив, или таблицу с прямой адресацией, T [0.. m – 1], каждая позиция, или ячейка (position, slot), которого соответствует ключу из множества **U**.

Таблицы с прямой адресацией



Реализация словарных операций:

DlRECT_ADDRESS_SEARCH (T, k)

return $T[k]$

DIRECT_ADDRESS_INSERT (T, x)

$T[\kappa ey[x]]$ ß x

DlRECT_ADDRESS_DELETE (T, x)

$T[\kappa ey[x]]$ ß NULL

Хеш-таблицы

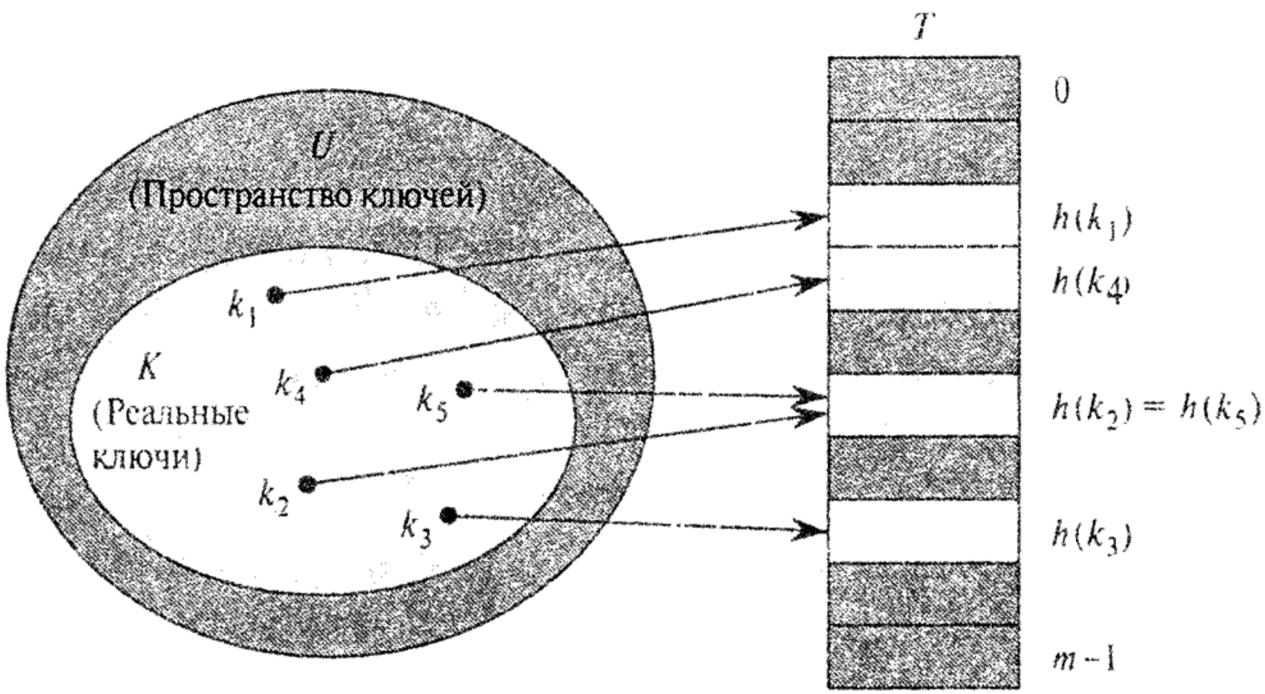
Недостаток прямой адресации очевиден: если пространство ключей **U** велико, хранение таблицы T размером |**U**| непрактично, а то и вовсе невозможно – в зависимости от количества доступной памяти и размера пространства ключей.

Множество **K** реально сохраненных ключей может быть мало по сравнению с пространством ключей **U**, а в этом случае память, выделенная для таблицы **T**, в основном расходуется напрасно.

Хеш-функция $h(k)$ для вычисления ячейки для данного ключа k.

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

Хеш-таблицы



Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо |**U**| значений мы можем обойтись всего лишь *m* значениями.

Хеш-таблицы

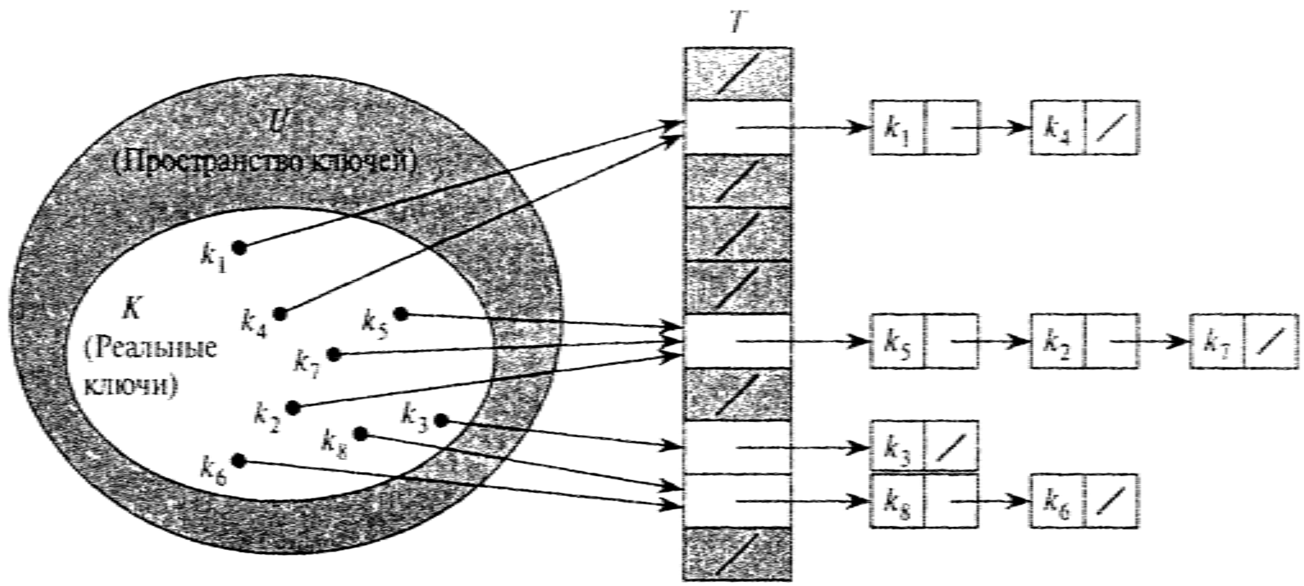
Коллизия – событие, когда два различных ключа хешированы в одну и ту же ячейку.

Полное разрешение коллизий невозможно, т.к. поскольку $|\mathbf{U}| > m$, должно существовать как минимум два ключа, которые имеют одинаковое хеш-значение. Хорошая хеш-функция в состоянии только минимизировать количество коллизий.

Аналоги:

1. В класса 32 человека. Хотя бы у двоих человек совпадает число в дне рождения.
2. Детская игра со стульями.

Разрешение коллизий при помощи цепочек



CHAINED_HASH_INSERT (T, x)

Вставить x в заголовок списка $T[h(key[x])]$

CHAINED_HASH_SEARCH (T, k)

Поиск элемента с ключом k в списке $T[h(k)]$

CHAINED_HASH_DELETE (T, x)

Удаление x из списка $T[h(key[x])]$

Разрешение коллизий при помощи цепочек

Пусть имеется хеш-таблица *T* с *m* ячейками, в которых хранятся *n* элементов.

Коэффициент заполнения таблицы T как $\alpha = n / m$, т.е. как среднее количество элементов, хранящихся в одной цепочке.

Средняя производительность хеширования зависит от того, насколько хорошо хеш-функция *h* распределяет множество сохраняемых ключей по *m* ячейкам в среднем. Будем полагать, что все элементы хешируются по ячейкам равномерно и независимо, и назовем данное предположение «*простым равномерным хешированием*» (*simple uniform hashing*).

Разрешение коллизий при помощи цепочек

Рассмотрим среднее количество элементов, которое должно быть проверено алгоритмом поиска. Необходимо рассмотреть два случая:

1. Поиск неудачен и в таблице нет элементов с ключом *k*.
2. Поиск заканчивается успешно и в таблице определяется элемент с ключом *k*.

Теорема 3.1. В хеш-таблице с разрешением коллизий методом цепочек среднее время неудачного поиска в предположении простого равномерного хеширования равно $\Theta(1+a)$

Теорема 3.2. В хеш-таблице с разрешением коллизий методом цепочек среднее время успешного поиска в предположении простого равномерного хеширования равно $\Theta(1+a)$

Хеш-функции. Качество хеш-функции

Предположение простого равномерного хеширования:

Для каждого ключа равновероятно помещение в любую из *m* ячеек, независимо от хеширования остальных ключей.

Хорошая хеш-функция

- Должна минимизировать шансы попадания близких в некотором смысле идентификаторов в одну ячейку хеш-таблицы.
- Не должна коррелировать с закономерностями, которым могут подчиняться существующие данные.

Построение хеш-функции методом деления

Построение хеш-функции *методом деления* состоит в отображении ключа k в одну из ячеек путем получения остатка от деления k на m, т.е. хеш-функция имеет вид $h(k) = k \bmod m$.

«Плохие» значения m:

1. Вида 2^p
2. Вида $2p – 1$

Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки.

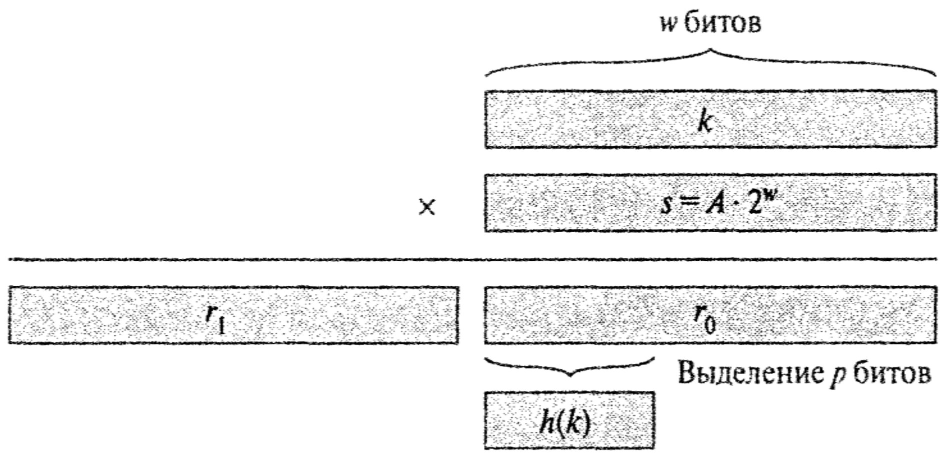
Построение хеш-функции методом умножения

Построение хеш-функции *методом умножения* выполняется в два этапа:

- Умножается ключ k на константу $0 < A < 1$ и получают дробную часть полученного произведения.
- Полученное значение умножается на m и для результата умножения вычисляется ближайшее целое число снизу, т.е.

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

Построение хеш-функции методом умножения



Значение *m* перестает быть критичным. Обычно величина *m* из соображений удобства реализации функции выбирается равной степени 2.

4. Открытая адресация. Реализация и анализ

Открытая адресация

При использовании метода *открытой адресации* все элементы хранятся непосредственно в хеш-таблице.

Поиск элемента – систематическая проверка ячеек таблицы до тех пор, пока не будет найден искомый элемент или пока не будет сделан вывод, что такого элемента в таблице нет.

Хеш-таблица может оказаться заполненной, делая невозможной вставку новых элементов; коэффициент заполнения a не может превышать 1.

Открытая адресация

Преимущество: Позволяет полностью отказаться от указателей, что уменьшает требования к памяти.

Для выполнения вставки необходимо последовательно *проверить*, или *исследовать* (*probe*), ячейки хеш-таблицы до тех пор, пока не будет найдена пустая ячейка, в которую помещается вставляемый ключ.

Вместо фиксированного порядка исследования ячеек $0, 1, \ldots, m - 1$ (для чего требуется $\Theta(n)$ времени), последовательность исследуемых ячеек зависит от вставляемого в таблицу ключа.

Открытая адресация

Расширим хеш-функцию, включив в нее в качестве второго аргумента номер исследования (начинающийся с 0)

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

Требование:

последовательность исследований

$$\langle h(k,0), h(k,1),\ldots, h(k, m-1) \rangle$$

должна представлять собой перестановку множества

$\{0,1,\ldots, m-1\}$

Открытая адресация

```
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
```

Алгоритм поиска ключа *k* исследует ту же последовательность ячеек, что и алгоритм вставки ключа *k*.

Открытая адресация

При удалении ключа из ячейки *i* мы не можем просто пометить ее значением NULL. Иначе будут «потеряны» ячейки, которые исследуются после *i*.

Одно из решений состоит в том, чтобы помечать такие ячейки специальным значением DELETED вместо NULL.

Упражнение:

Измените процедуру HASH_INSERT, чтобы она работала с ячейками со значением DELETED.

Открытая адресация

Предположение равномерного хеширования:

Для каждого ключа в качестве последовательности исследований равновероятны все $m!$ перестановок множества $\{0,1,\ldots, m-1\}$

Распространенные методы для вычисления последовательности исследований (не удовлетворяют *Предположению*):

- линейное исследование,
- квадратичное исследование,
- двойное хеширование (наилучший из трех рассматриваемых)

Открытая адресация

Вспомогательная хеш-функция (*auxiliary hash function*):

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

Метод *линейного исследования*:

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

Порядок исследования ячеек:

$$T[h'(k)],\ \ T[h'(k)+1],\ \ T[h'(k)+2],\ldots$$

Проблема первичной кластеризации.
Вероятность заполнения пустой ячейки, которой предшествуют *i* заполненных ячеек, равна $(i+1)/m$.

Квадратичное исследование

Квадратичное исследование использует хеш-функцию вида

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

Порядок исследования ячеек:

$$T[h'(k)],\ \ T[h'(k) + c_1 + c_2],\ \ T[h'(k) + 2c_1 + 4c_1], \ldots$$

Работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c_1, c_2 и m.

Более мягкая вторичная кластеризация из-за равенства хеш-функций различных ключей при одинаковых начальных позициях.

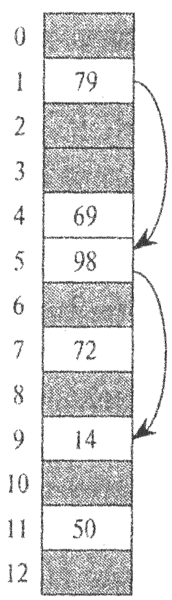
Двойное хеширование

Двойное хеширование использует хеш-функцию вида:

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

Порядок исследования ячеек:

$$T[h_1(k)],\ T[h_1(k)+h_2(k)],\ T[h_1(k)+2h_2(k)],\ldots$$



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

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

Двойное хеширование

Значение $h_2(k)$ должно быть взаимно простым с размером хеш-таблицы *m.*

Примеры вариантов:

1. $m = 2^p$, h_2 возвращает нечетные числа.
2. m – простое число, h_2 возвращает натуральные числа, меньшие m.

$$h_1(k) = k \bmod m,$$
$$h_2(k) = 1 + (k \bmod m'),$$

где m' немного меньше m.

Производительность двойного хеширования достаточно близка к производительности «идеальной» схемы равномерного хеширования.

Анализ хеширования с открытой адресацией

Используется коэффициент заполнения $a = n / m$ хеш-таблицы при n и m, стремящихся к бесконечности.

Будем считать, что используется равномерное хеширование, т.е. последовательность исследований $\langle h(k,0), h(k,1),\ldots, h(k,m-1)\rangle$ является одной из возможных перестановок $\langle 0,1,\ldots,m-1\rangle$.

Теорема 4.1. Среднее количество исследований при неуспешном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $a = n / m < 1$ в предположении равномерного хеширования не превышает .

Анализ хеширования с открытой адресацией

Следствие 4.1. Вставка элемента в хеш-таблицу с открытой адресацией и коэффициентом заполнения a в предположении равномерного хеширования, требует в среднем не более 1/(1- a) исследований.

Теорема 4.2. Математическое ожидание количества исследований при удачном поиске в хеш-таблице с открытой адресацией и коэффициентом заполнения $a < 1$, в предположении равномерного хеширования и равновероятного поиска любого из ключей, не превышает

$$\frac{1}{a} \ln \frac{1}{1-a}$$

5. Список источников

1. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2005. – 1296 с. : ил. – Парал. тит. англ.
2. Макконнелл Дж. Основы современных алгоритмов. 2-е дополненное издание. – Москва: Техносфера, 2004. – 368с.