6. Динамические структуры данных. Примеры работы со стеком.

1.1. Стеки

Стек — частный случай списка. Стек удовлетворяет принципу LIFO — «Last In First Out» (последний зашел — первый вышел). На основе стеков устроены большинство компьютерных операций, в частности, рекурсивные функции основаны на стеках.

По своему устройству стек напоминает детскую игрушку — пирамидку. На примере пирамидки понятно, что для того, чтобы добраться до одетого первым элемента, необходимо снять все верхние элементы. Другой пример стека — имеется сосуд, который последовательно заполняется шарами. Тогда, чтобы вынуть последний шар из сосуда, необходимо вынуть все остальные.

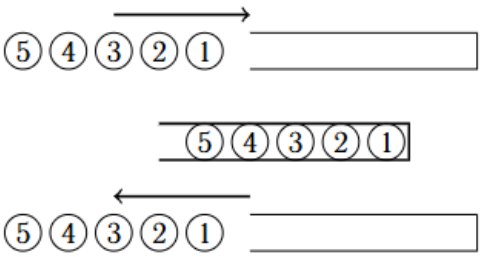


Рис. 1.2. Представление стека

Точно также для стека определены только две операции: добавить элемент в начало стека и извлечь элемент из начала стека. Других операций для стека НЕ ОПРЕДЕЛЕНО. Как

```
struct stack{
  int inf;
  stack *next;
};
```

```
void push(stack *&h, int x){
  stack *r = new stack; //создаем новый элемент
  r->inf = x;           //поле inf = x
  r->next = h;          //следующим элементов является h
  h = r;                //теперь r является головой
}
```

```
void reverse(stack *&h){
  stack *head1 = NULL;
  while (h)
    push(head1, pop(h));
  h = head1;
}
```

```
int pop (stack *&h){
  int i = h->inf;   //значение первого элемента
  stack *r = h;     //указатель на голову стека
  h = h->next;      //переносим указатель на следующий элемент
  delete r;         //удаляем первый элемент
  return i;         //возвращаем значение
}
```

7. Динамические структуры данных. Пример работы с очередью.

1.2. Очередь

Очередь также представляет собой частный случай списка. Элементы очереди устроены по принципу FIFO — First In First Out (Первый зашел — первый вышел). Очередь можно представить как сосуд без дна. С одной стороны заполняется, с другой извлекается:

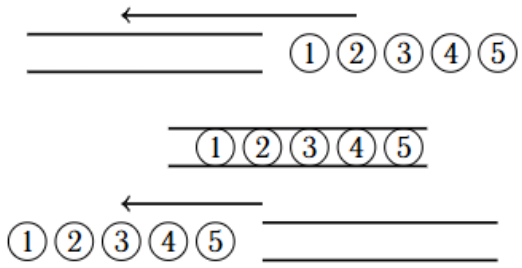


Рис. 1.3. Представление очереди

Как видно из рисунка **??** вставлять или удалять элементы в середину очереди нельзя. То есть, для очереди определены только две операции: добавить элемент в хвост очереди и извлечь элемент из начала очереди. Других операций для стека НЕ ОПРЕДЕЛЕНО.

```
void push (queue *&h, *&t, int x){ //вставка элемента в очередь
    queue *r = new queue;       //создаем новый элемент
    r->inf = x;
```



```
    r->next = NULL;             //всегда последний
    if (!h && !t){               //если очередь пуста
      h = t = r;                 //это и голова и хвост
    }
    else {
      t->next = r;              //r - следующий для хвоста
      t = r;                    //теперь r - хвост
    }
}
```

```
struct queue {
   int inf;
   queue *next;
};
```

```
int pop (queue *&h, *&t){ //удаление элемента из очереди
    queue *r = h;         //создаем указатель на голову
    int i = h->inf;       //сохраняем значение головы
    h = h->next;          //сдвигаем указатель на следующий элемент
    if (!h)               //если удаляем последний элемент из очереди
      t = NULL;
    delete r;             //удаляем первый элемент
```

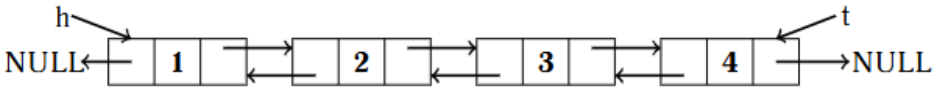


```
    return i;
    }
}
```

8. Динамические структуры данных. Двусвязный список. Поиск элемента. Вставка элемента в двусвязный список.

1.3. Двусвязный список

Наиболее общий случай связных списков. Каждый элемент списка состоит из трех полей: информационного и двух ссылочных на следующий и предыдущий элементы:



С элементами списка можно выполнять любые действия: добавлять, удалять, просматривать и т. д. Вставка и удаление элемента выполняется за время $O(1)$, так как надо всего лишь поменять значения в ссылочных полях.

Список относится к элементам с последовательным доступом, т. е., чтобы просмотреть пятый элемент, необходимо последовательно просмотреть первые четыре элемента. Для перехода к следующему элементу указателю `p` присваивается значение `p->next`. Так как список двусвязный, можно просматривать элементы как в прямом (`p = p->next`), так и в обратном порядке (`p = p->prev`).

```
struct list {
    int inf;
    list *next;
    list *prev;
};
```

```
void push (list *&h, list *&t, int x){ //вставка элемента в конец списка
    list *r = new list;                //создаем новый элемент
    r->inf = x;
    r->next = NULL;                    //всегда последний
    if (!h && !t){                     //если список пуст
      r->prev = NULL;                  //первый элемент
      h = r;                           //это голова
     }
    else{
      t->next = r;                     //r - следующий для хвоста
      r->prev = t;                     //хвост - предыдущий для r
    }
    t = r;                             //r теперь хвост
}
```

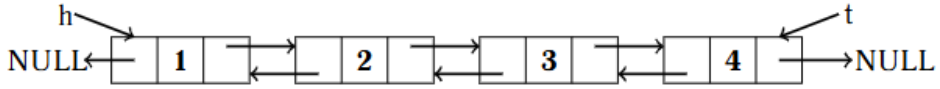
```
list *find ( list *h, list *t, int x){ //печать элементов списка
   list *p = h;                //укзатель на голову
   while (p){                  //пока не дошли до конца списка
     if (p->inf == x) break // если нашли, прекращаем цикл
     p = p->next;              //переход к следующему элементу
   }
   return p;                   //возвращаем указатель
}

void insert_after ( list *&h, list *&t, list *r, int y){ //вставляем после r
   list *p = new list;                  //создаем новый элемент
   p->inf = y;
   if (r == t){                         //если вставляем после хвоста
     p->next = NULL;                    //вставляемый эл-т - последний
     p->prev = r;                       //вставляем после r
     r->next = p;
     t = p;                             //теперь хвост - p
   }
   else{                                //вставляем в середину списка
      r->next->prev = p;                //для следующего за r эл-та предыдущий - p
      p->next = r->next;                //следующий за p - следующий за r
      p->prev = r;                      //p вставляем после r
      r->next = p;
   }
}
```

9. Динамические структуры данных. Двусвязный список. Поиск элемента. Удаление элемента из двусвязного списка.

1.3. Двусвязный список

Наиболее общий случай связных списков. Каждый элемент списка состоит из трех полей: информационного и двух ссылочных на следующий и предыдущий элементы:



С элементами списка можно выполнять любые действия: добавлять, удалять, просматривать и т. д. Вставка и удаление элемента выполняется за время $O(1)$, так как надо всего лишь поменять значения в ссылочных полях.

Список относится к элементам с последовательным доступом, т. е., чтобы просмотреть пятый элемент, необходимо последовательно просмотреть первые четыре элемента. Для перехода к следующему элементу указателю `p` присваивается значение `p->next`. Так как список двусвязный, можно просматривать элементы как в прямом (`p = p->next`), так и в обратном порядке (`p = p->prev`).

```
list *find (list *h, list *t, int x){ //печать элементов списка
   list *p = h;                  //укзатель на голову
   while (p){                    //пока не дошли до конца списка
     if (p->inf == x) break // если нашли, прекращаем цикл
     p = p->next;                //переход к следующему элементу
   }
   return p;                     //возвращаем указатель
}

void del_node (list *&h, list *&t, list *r){ //удаляем после r
   if (r == h && r == t)                  //единственный элемент списка
     h = t = NULL;
   else if (r == h){                      //удаляем голову списка
     h = h->next;                         //сдвигаем голову
     h->prev = NULL;
```



```
   }
   else if (r == t){                      //удаляем хвост списка
     t = t->prev;                         //сдвигаем хвост
     t->next = NULL;
   }
   else{
     r->next->prev = r->prev;             //для следующего от r предыдущим
          становится r->prev
     r->prev->next = r->next;             //для предыдущего от r следующим
          становится r->next
   }
   delete r;                              //удаляем r
}
```

10. Деревья. Основные понятия. Бинарные деревья. Обходы бинарных деревьев.

Дерево — иерархическая абстрактная структура данных. Используется в различных областях.

Формально,

Определение 1. Дерево — набор элементов, связанных отношениями «родитель – ребенок», удовлетворяющих следующим условиям:

1. Если дерево непустое, то существует вершина, называемая **корнем дерева**, не имеющая родителя.
2. Каждый узел v дерева имеет одного родителя w. Тогда v является ребенком w.

Дерево можно также определить рекурсивно:

Определение 2. Дерево — набор элементов, удовлетворяющих следующим условиям:

1. Если дерево непустое, то существует вершина, называемая **корнем дерева**, не имеющая родителя.
2. Каждый узел v можно трактовать как корень своего дерева. Такие деревья называют **поддеревом**.

Визуально, дерево представляется в виде узлов и ребер, соединяющих родителя и его детей. Не бывает изолированных узлов. Если два узла имеют одного родителя, то эти узлы не могут быть соединены ребром (тогда это будет граф.) В английских источниках узлы, имеющие одного родителя, называются **siblings**. В русском языке аналога этого слова нет (только в биологии встречается сибс.)

Родитель и ребенок — это пара смежных узлов. Если говорить о более дальних связях, то вводится понятие **пути**. Путь из узла A до узла B — это набор узлов $A, a_1, \ldots, a_n, B$ таких, что узел a_{i+1} является ребенком узла a_i. **Длина пути** — число таких узлов минус единица.

Если существует путь из узла A в узел B, тогда узел A является **предком** узла B, а узел B — **потомком** узла A. Корень дерева является предком всех остальных узлов.

Узлы разделяются на **внутренние** и **внешние**. Внешние узлы не имеют детей, очень часто называются также **листьями**.

Введем понятие **высоты узла** — максимальная длина пути от узла до листа. (Находим длины путей от узла до всех листьев и выбираем максимальную из них). Соответственно, **высота дерева** — максимальная длина пути от корня до листа.

Глубина узла — длина пути от корня до узла. Глубина узла находится однозначно, так как у каждого узла только один родитель.

Часто говорят, что узлы расположены по уровням. По умолчанию корень расположен на нулевом уровне. Дети корня — на первом, «внуки» — на втором и т. д.

Деревья бывают **упорядоченными** и **неупорядоченными**. В случае упорядоченного дерева является важным порядок следования узлов на уровне, обычно следование идет слева направо. Это определяется решаемой задачей, т. е., в случае упорядоченного дерева, деревья, изображенные на рисунке 1.2а) и 1.2б) будут различными, в случае неупорядоченного — нет.

1.1. Бинарное дерево

В общем случае, родитель может иметь любое количество детей, но использование такой структуры затруднено. Значительно удобнее использовать **бинарное дерево**, т. е.,

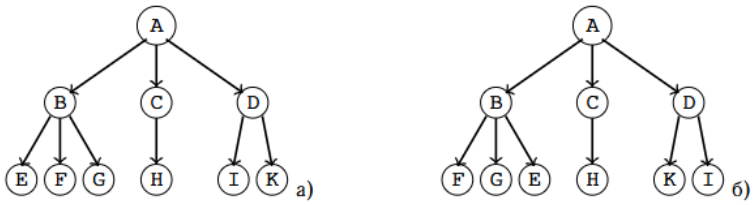


Рис. 1.2. Пример дерева (а) и (б)

дерево, каждый узел которого имеет не более двух детей. Будем считать, что бинарное дерево является упорядоченным. В дальнейшем будем рассматривать только бинарное дерево.

```
struct tree
{
    Item inf;
    tree *left;
    tree *right;
    tree *parent;
};
```

```
tree *node(Item x){
  tree *n = new tree;
  n->inf = x;
  n->parent = NULL;
  n->right = NULL;
  n->left = NULL;
}
```

1.2. Обходы

Построенное дерево необходимо вывести на экран. Для этого используются обходы деревьев.

Обход — способ вывода всех узлов дерева ровно один раз. Поскольку бинарное дерево состоит из левого поддерева (L), правого поддерева (R), корня (N). Понятно, что существует шесть вариантов обходов: LRV, LVR, VLR, RLV, RVL, VRL. Последние три являются симметричными первым трем, поэтому обычно рассматриваются три обхода:

- **Обратный**:
  1. Посетили левое поддерево;
  2. Посетили правое поддерево;
  3. Посетили корень.

- **Прямой**:
  1. Посетили корень;
  2. Посетили левое поддерево;
  3. Посетили правое поддерево.

- **Симметричный**:
  1. Посетили левое поддерево;
  2. Посетили корень;
  3. Посетили правое поддерево.

```
void preorder (tree *tr){ // прямой обход (К-Л-П)
```



```
  if (tr){
    cout << tr->inf;      //корень
    preorder(tr->left);   //левое
    preorder(tr->right); //правое
  }
}

void postorder (tree *tr){ // обратный обход (Л-П-К)
  if (tr){
    postorder(tr->left);   //левое
    postorder(tr->right); //правое
    cout << tr->inf;      //корень
  }
}

void inorder (tree *tr){ // симметричный обход (Л-К-П)
  if (tr){
    inorder(tr->left);   //левое
    cout << tr->inf;      //корень
    inorder(tr->right); //правое
  }
}
```

11. Работа с деревом бинарного поиска.

1.4. Дерево бинарного поиска

Рассмотрим еще один случай бинарного дерева: дерево бинарного поиска. В этом случае добавляется дополнительное условие на узлы: для любого узла левый ребенок меньше своего родителя, правый — больше. В случае случайного распределения данных такое дерево подходит для поиска данных, так как необходимо пройти только по одной ветке дерева. Но, можно подобрать данные таким образом, что дерево будет представлять собой одну ветку, что увеличивает поиск до $O(n)$, где n — это количество элементов в дереве.

Для обхода дерева удобно использовать симметричный обход, так как в этом случае на экран будет выведена отсортированная последовательность.

Поскольку неравенства строгие, то дерево не содержит повторяющихся элементов, при вставке в дерево они просто игнорируются.

Простейшая реализация дерева бинарного поиска состоит в следующем:

1. Первый элемент всегда является корнем;

2. Если вставляемый элемент меньше корня, ищем подходящее место на левой ветке;
3. Если вставляемый элемент больше корня — на правой.

Поиск элемента

- Если указатель равен `NULL`, значит элемента в дереве нет, возвращаем `NULL`;
- Если значение текущего элемента равно искомому значению, возвращаем указатель на этот элемент;

- Если значение текущего элемента больше искомого, рекурсивно вызываем поиск по левой ветке;
- Иначе рекурсивно вызываем поиск по правой ветке.

Поиск минимального элемента

- Если нет левого ребенка, элемент минимальный и возвращаем указатель на данный элемент.
- Иначе рекурсивно вызываем функцию по левой ветке.

Поиск максимального элемента

- Если нет правого ребенка, элемент максимальный и возвращаем указатель на данный элемент.
- Иначе рекурсивно вызываем функцию по правой ветке.

Поиск следующего элемента

- Если существует правый ребенок, то ищем минимальный по правой ветке.
- Иначе, идем вверх по дереву, до тех пока не дойдем до корня или пока текущий элемент остается правым ребенком. Возвращаем указатель на родителя.

Поиск предыдущего элемента

- Если существует левый ребенок, то ищем максимальный по левой ветке.
- Иначе, идем вверх по дереву, до тех пока не дойдем до корня или пока текущий элемент остается левым ребенком. Возвращаем указатель на родителя.

12. Работа с идеально сбалансированным деревом.

1.5. Идеально сбалансированное дерево

Дерево бинарного поиска обладает плохим свойством, что при определенном наборе данных (например, отсортированном) может обладать только одной веткой и, соответственно, его высота будет равна $N - 1$.

Рассмотрим сначала *идеально сбалансированное дерево* — дерево, для каждого узла которого число потомков левого узла отличается от числа потомков правого узла не более чем на единицу. При этом пока уберем требование бинарного поиска (значения узлов не важно).

Для построения идеально сбалансированного дерева необходимо заранее знать об-

щее количество элементов. Тогда дерево строится по следующему алгоритму:

- Пусть дано n элементов. Первый элемент списка является корнем.
- В левом поддереве будет $\frac{N}{2}$ элементов, в правом — $\frac{N}{2} - 1$ элемент (общее число элементов минус левое поддерево минус корень.)
- Сначала рекурсивно заполняем левую ветку, потом правую.

Например, дано 10 элементов: 4 5 3 7 8 6 9 1 2 0.

- Корень — 4. Левое поддерево содержит 5 узлов: 5 3 7 8 6. Правое — 4 узла: 9 1 2 0.
- Узел — 5. Левое поддерево содержит 2 узла: 3 7. Правое — 2 узла: 8 6.
- Узел — 3. Левое поддерево содержит 1 узел: 7. Правое — 0 узлов.
- Узел — 7. Это лист.
- Узел — 8. Левое поддерево содержит 1 узел: 6. Правое — 0 узлов.
- Узел — 6. Это лист. Левое поддерево для корня построено, переходим к правому.
- Узел — 9. Левое поддерево содержит 2 узла: 1 2. Правое — 1 узел: 0.
- Узел — 1. Левое поддерево содержит 1 узел: 2. Правое — 0 узлов.
- Узел — 2. Это лист.
- Узел — 0. Это лист. Дерево построено.

Можно:
1. Добавлять элемент
2. Удалять элемент
3. Искать элемент через обход
4. Вывод с помощью обхода

* чтобы не занимать много места страницами кода

13. Графы. Основные понятия.

1. **Граф**
   - Формально: $G = (X, A)$, где:
     - X — множество вершин (узлов)
     - A — множество рёбер (дуг для ориентированных графов)
   - Типы:
     - **Ориентированный (орграф)**: рёбра имеют направление (дуги).
     - **Неориентированный**: рёбра без направления.
     - **Взвешенный**: каждому ребру присвоен вес (например, расстояние).
2. **Основные элементы**
   - **Вершина (узел)**: элементарная единица графа.
   - **Ребро**: связь между двумя вершинами. Для орграфа — дуга (упорядоченная пара).
   - **Степень вершины**:
     - В неориентированном графе: количество инцидентных рёбер.
     - В орграфе: полустепень исхода (исходящие дуги) и захода (входящие дуги).
3. **Виды графов**
   - **Полный граф**: все вершины соединены рёбрами. Число рёбер: $\frac{N(N-1)}{2}$ для неориентированного графа.
   - **Планарный граф**: можно изобразить на плоскости без пересечений рёбер.
   - **Регулярный граф**: все вершины имеют одинаковую степень.
   - **Простой граф**: без петель и кратных рёбер.
4. **Связность**
   - **Связный граф**: существует путь между любыми двумя вершинами.
   - **Компонента связности**: максимальный связный подграф.
   - **Мост**: ребро, удаление которого увеличивает число компонент связности.
   - **Точка сочленения**: вершина, удаление которой увеличивает число компонент.
5. **Пути и циклы**
   - **Путь**: последовательность вершин $a_0 \to a_1 \to \cdots \to a_n$, где каждые две соседние вершины соединены ребром.
   - **Цикл**: путь, начинающийся и заканчивающийся в одной вершине (длина ≥ 3).
   - **Эйлеров цикл**: проходит по каждому ребру ровно один раз. Условие: все степени вершин чётные.
   - **Гамильтонов цикл**: проходит через каждую вершину ровно один раз (NP-полная задача).
6. **Представление графов**
   - **Матрица смежности**: квадратная матрица $N \times N$, где $a[i][j] = 1$, если есть ребро $i \to j$.
   - **Список смежности**: для каждой вершины хранится список смежных вершин. Эффективен для разреженных графов.
7. **Обходы графа**
   - **В глубину (DFS)**: рекурсивный обход с погружением "вглубь".
   - **В ширину (BFS)**: обход по уровням с использованием очереди.
8. **Алгоритмы**
   - **Топологическая сортировка**: линейное упорядочение вершин орграфа без циклов.
   - **Поиск компонент сильной связности**: для орграфа (алгоритм Косарайю).
   - **Алгоритм Флойда-Уоршелла**: поиск кратчайших путей между всеми вершинами.

14. Графы. Обходы в глубину и ширину.

1. Обход в глубину (Depth-First Search, DFS)

Принцип работы:

Рекурсивно исследуется каждая ветвь графа до конца, затем происходит возврат (backtracking) к ближайшей непосещённой вершине.

Алгоритм:

1. Пометить текущую вершину как посещённую.
2. Для всех смежных вершин:
    - Если вершина не посещена, рекурсивно вызвать DFS для неё.

Пример для графа:

text Copy Download

```
    0
  / | \
 1  2  3
   / \
  4   5
```

DFS-порядок (старт с 0):

`0 → 1 → 2 → 4 → 5 → 3`

2. Обход в ширину (Breadth-First Search, BFS)

Принцип работы:

Посещение вершин "по уровням": сначала все вершины на расстоянии 1, затем на расстоянии 2 и т.д. Использует очередь.

Алгоритм:

1. Поместить стартовую вершину в очередь.
2. Пока очередь не пуста:
    - Извлечь вершину из очереди.
    - Посетить её.
    - Добавить в очередь все непосещённые смежные вершины.

Пример для того же графа:

BFS-порядок (старт с 0):

`0 → 1 → 2 → 3 → 4 → 5`

15. Библиотека STL. Контейнеры.

2. Контейнеры

Контейнеры делятся на две больших группы: *последовательные* и *ассоциативные*.

Последовательные контейнеры представляют собой набор элементов, расположение которых в контейнеры зависит от порядка поступления в контейнер: добавление элемента происходит либо в конец контейнера, либо в начало. К последовательным контейнерам относятся `vector, list, deque`.

Ассоциативные контейнеры представляют собой отсортированную последовательность, расположение элемента зависит от его значения. К ассоциативным контейнерам относятся `set, multiset, map, multimap`.

Для контейнеров должны выполняться три основных требования:

1. Поддерживается семантика значений вместо ссылочной семантики. При вставке элемента контейнер создает его внутреннюю копию, а не сохраняет ссылку на объект.

2. Элементы в контейнере располагаются в определенном порядке. При повторном переборе порядок должен остаться прежним. Для этого определены возвращающие итераторы для каждого контейнера.

3. В общем случае операции с контейнерами небезопасны. Необходимо следить за выполнением операций.

Для использования контейнеров необходимо подключать соответствующие библиотеки. Например, для использования списка подключается библиотека `#include<list>` и т. д.

Описание любого контейнера: `cont<type> x;`, где `cont` — наименование контейнера, `type` — тип элементов (может быть любым, включая контейнеры). Например, `vector<int> c`.

Основные методы, определенные для всех контейнеров:

- `x.size()` — возвращает размер контейнера.
- `x.empty()` — возвращает `true`, если контейнер пустой.
- `x.insert(pos, elem)` — вставляет копию `elem` в позицию `pos`. Возвращаемое значение зависит от контейнера.
- `x.erase(beg,end)` — удаляет все элементы из диапазона $[beg, end)$.
- `x.clear()` — удаляет из контейнера все элементы.

2.6. Создание контейнеров

Существует несколько возможностей создания контейнеров.

- Создание пустого контейнера.

  Например, `vector<int> c`

- Создание контейнера размеров n, заполненного элементами по умолчанию (0 — для целых чисел, 0.0 — для вещественных и т. д.).

  Например, `vector<int> c(n)`

- Создание копии контейнера того же типа.

  Например, `vector<int> c(c1)`

- Создание контейнера и инициализация его копиями всех элементов в интервале $[beg, end)$.

  Этот способ позволяет переписывать элементы из одного контейнера в другой. Например, сначала создать список, заполнить его элементами, потом создать копию этого списка в виде множества и использовать поиск уже в множестве.

  ```
  list<int>a(n);
  ```

  Заполнили список. Создали множество, заполнив его копиями этого списка.

  ```
  set<int>b(a.begin(), a.end());
  ```

16. Библиотека STL. Итераторы.

3. Итераторы

Итератор — это объект, предназначенный для перебора элементов контейнера STL. Итератор представляет некоторую позицию в контейнере.

Основные операторы:

- $*$ — получение значения элемента в текущей позиции итератора (`*iter`). Для `map` необходимо использовать $->$: `iter->first, iter->second`.

- $++$ — перемещение итератора к следующему элементу контейнера. Для разных контейнеров имеет разный смысл:

  Вектор, дек — переход к следующему элементу, увеличивая адрес на `sizeof()` байт.

  Список — переход к следующему элементу по полю `p->next`

  Множество и отображение — переход к следующему элементу, используя симметричный обход дерева.

- $==$ и $!=$ — проверка совпадений позиций, представленных двумя итераторами. Для вектора и дека определены операции сравнения $<$ и $>$.

17. Библиотека STL. Алгоритмы.
4. Алгоритмы в библиотеке STL

Для работы с встроенными алгоритмами необходимо подключить библиотеку `<algorithm>`. Некоторые алгоритмы, необходимые для обработки числовых данных, определены в файла `<numeric>`.

Подробно описание алгоритмов рассматривать не будем. Остановимся только на общих принципах.

---

1. Большая часть алгоритмов возвращает итераторы. Большая часть алгоритмов в качестве параметров использует итераторы. Например, `vector<int>::iterator iter = min_element(x.begin(), x.end())`в качестве параметров использует итераторы. Результатом будет итератор, указывающий на элемент с минимальным значением.

   Очень опасно использовать, например, следующую запись: `remove(x.begin(), x.end(), *iter)`. Предполагается, что с помощью этого алгоритма удаляются все минимальные элементы. Но на самом деле удаляется только элемент, на который указывает `iter`.

   Для правильной работы необходимо выполнить:

   - `vector<int>::iterator iter = min_element(x.begin(), x.end())`
   - `int Min = *iter`
   - `remove(x.begin(), x.end(), Min)`

2. При работе с интервалами всегда подразумевается полуоткрытый интервал $[beg, end)$, т. е., включая beg и не включая end. Естественно, что $beg \leqslant end$.

   Если алгоритм использует несколько интервалов, то для первого задается начало и конец интервала, а для второго — только начало. Например, алгоритм `equal(x.begin(), x.end(), y.end())` сравнивает поэлементно содержимое коллекции x с содержимым коллекции y.

   ВАЖНОЕ ТРЕБОВАНИЕ для алгоритмов, осуществляющих запись в контейнеры или для алгоритмов, использующих несколько интервалов, необходимо заранее убедится, что размер контейнера достаточен для работы с алгоритмом.

3. Алгоритмы, предназначенные для «удаления» элементов (`remove` и `unique`), на самом деле просто переставляют элементы, «удаляя» нужные, но не изменяет размер контейнера. Однако эти алгоритмы возвращают итератор, указывающий на новый конец контейнера. И можно удалить все элементы, расположенные после этого итератора с помощью метода `x.erase()`:

   - `vector<int>::iterator iter = remove(x.begin(), x.end(), val)`
   - `x.erase(iter, x.end())`

4. Модифицирующие алгоритмы (алгоритмы, удаляющие элементы, изменяющие порядок их следования или значения) не могут применяться для ассоциативных контейнеров.

Некоторые алгоритмы имеют могут быть представлены в нескольких видах: обычном, с суффиксом `_if` и суффиксом `_copy`.

- Обычный алгоритм используется при передаче значения. Например, `replace(x.begin(), x.end(), old, New)`

  заменяет все элементы со значением `old` значением `New`.

- Алгоритм с суффиксом `_if` используется при передаче функции. Например,

  `replace_if(x.begin(), x.end(), func, New)`

  заменяет все элементы, для которых `func(elem)` возвращает `true`, значением `New`.

- Алгоритм с суффиксом `_copy` используется в случае, когда результат записывается в новый контейнер (или интервал). Необходимо убедиться, что памяти в приемном контейнере хватает для копирования результата. Например,

  `replace_copy(x.begin(), x.end(), y.begin(), old, New)`

  заменяет все элементы со значением `old` значением `New` и копирует результат в контейнер `y`.

Полный список всех алгоритмов можно найти в справочной литературе по библиотеке STL.

18. . Хэширование. Методы разрешения коллизий
Хэширование

Поиск с помощью дерева бинарного поиска в лучшем случае занимает время $O(N \log N)$, в худшем — $O(N)$. Но само построение сбалансированного дерева бинарного поиска достаточно сложный процесс и не всегда удобный.

Рассмотрим другой способ построения базы для поиска — хэширование. Вообще, хэширование достаточно распространено. Например, во многих базах для аутентификации используется не пароль, а хэш этого пароля. Т. е., Вы вводите пароль, вычисляется хэш этого пароля и проверяется с хранящимися данными.

В данном разделе не будут рассмотрены сложные функции хэширования. Рассмотрим только простейшие варианты.

Основная цель хэш-таблиц — расположить элементы в таблице по K ячейкам в соответствии со значением хэш-функции $h(x)$. Для каждого элемента x строится хэш-функция, такая, что значение $h(x)$ находится в интервале $[0, \ldots, B-1]$. Потом элемент x ставится в ячейку, соответствующую значению хэш-функции.

Элемент x часто называют ключом, $h(x)$ — хэш-значением. Значение хэш-функции должно быть обязательно целочисленным. В дальнейшем будет предполагать, что элементы располагаются по ячейкам хэш-функции равномерно и независимо.

Для простых хэш-функций достаточно часто бывают ситуации, когда несколько ключей имеют одинаковые значения хэш-функции. Такие ситуации называют *коллизиями*.

Предполагаем, что ключ всегда является целочисленным значением. Если, например, ключом является строка, то можно представить ее в виде целого числа, написанного в определенной системе счисления.

Например, строка `Hy.` В таблице ASCII кодов это слово представляется как (72, 121). Основание системы счисления— 128. Следовательно, строку можно представить как число $X = 72 \times 128 + 121 = 9337$.

Рассмотрим два метода разрешения коллизий: открытое хэширование (метод цепочек) и закрытое хэширование (метод открытой адресации).

1.1 Открытое хэширование

Представление данных в таком случае напоминает поразрядную сортировку (можно считать хэш-таблицей, где хэш-функция — цифра в определенном разряде).

Пусть есть N данных. Размер таблицы — M, следовательно, хэш-функция принимает значения в диапазоне $[0, \ldots, M-1]$.

Хэш-таблица представляет собой массив списков. Список выбран как структура, позволяющая удалять данные за константное время.

При хорошей хэш-функции в каждой ячейке таблицы будет находится в среднем $\alpha = \dfrac{N}{M}$. Назовем α *коэффициентом заполнения* хэш-таблицы.

Алгоритм 1: Создание хэш-таблицы

Вход: A — массив размерности N, M — размерность хэш-таблицы
Выход: Хэш-таблица
начало алгоритма
  цикл пока *не дошли до конца массива* **выполнять**
    · Определяем значение хэш-функции $k = h(A[i])$;
    · Добавляем элемент массива в k-ый список хэш-таблицы;
конец алгоритма

Например, для $N = 200$, $M = 50$ коэффициент заполнения $\alpha = 4$. Следовательно, поиск и удаление элемента занимает время $O(1 + \alpha)$.

Алгоритм 2: Поиск или удаление элемента хэш-таблицы

Вход: A — хэш-функция размерности M, X — элемент для поиска или удаления
Выход: Измененная хэш-таблица (при удалении) или указатель на найденный элемент (при поиске)
начало алгоритма
  · Определяем значение хэш-функции для элемента X;
  цикл пока *не дошли до конца списка соответствующей ячейки хэш-таблицы* **выполнять**
    · Ищем необходимый элемент;
    · Удаляем найденный элемент;
конец алгоритма

Основные достоинства открытого хэширования:

1. Неограниченный размер хэш-таблицы (элементы в списки можно добавлять без ограничений)
2. Поиск и удаление за время $O(1 + \alpha)$, что меньше поиска в сбалансированном дереве бинарного поиска.

1.1.1 Метод деления

Самая простая хэш-функция — остаток от деления на M :

$h(x) = x \mod M$.

Хэширование достаточно быстрое. Если правильно подобрать размер таблицы, то хэш-функция достаточно эффективна.

Нельзя выбирать в качестве M степень двойки. Неудачным является и выбор $M = 2^P - 1$.

Самым удачным является выбор в качестве M простого числа, достаточно далекого от степени двойки.

Например, пусть $N = 2000$. Предположим, что в данном случае достаточно, чтобы коэффициент заполнения таблицы был равен трем. Следовательно, $M \approx \frac{N}{\alpha} \approx 701$. Тогда для данного случая хэш-функция — $h(x) = x \mod 701$.

Пример 1.1. Дан набор чисел: 12, 17, 25, 41, 23, 11, 24, 21, 26, 44, 33, 10, 20, 19, 29. Построить хэш-таблицу.

$N = 15$.

Выбираем в качестве M простое число, например, 7. Выбор не самый удачный ($7 = 2^3 - 1$,) но для $N = 15$ в любом случае хэш-таблица не будет самой наглядной. Просто для примера.

Результат:

0:	21
1:	29
2:	23 44
3:	17 24 10
4:	25 11
5:	12 26 33 19
6:	41 20

1.1.2 Метод умножения

1. Сначала x умножается на коэффициент $0 < A < 1$ и получаем дробную часть полученного выражения.

2. Результат умножается на M и берется целая часть.

Таким образом хэш-функция имеет вид $h(x) = \lfloor M(xA \mod 1) \rfloor$, где $xA \mod 1 = (xA - \lfloor xA \rfloor)$ — получение дробной части, $\lfloor z \rfloor$ — целая часть числа z.

В качестве A выбирается золотое сечение: $A = \frac{\sqrt{5}-1}{2} \approx 0.61803\ldots$

В данном случае в качестве M как раз лучше всего выбрать степень двойки для удобства умножения.

Пример 1.2. Дан набор чисел: 12, 17, 25, 41, 23, 11, 24, 21, 26, 44, 33, 10, 20, 19, 29. Построить хэш-таблицу.

$N = 15$.

Выбираем в качестве M степень двойки, например, 8.

Рассмотрим на примере $x = 12$.

$A \times x = 12 * 0.618034 = 7.416408$. Дробная часть — 0.416408. Умножаем на $M = 8$ и берем целую часть. Следовательно, $h(12) = \lfloor 0.416408 \times 8 \rfloor = 3$.

Результат:

0:	26
1:	23 44 10
2:	41 20
3:	12 25 33
4:	17
5:	19
6:	11 24
7:	21 29

1.2 Закрытое хэширование

В случае закрытого хэширования все элементы располагаются непосредственно в таблице. Это позволяет избавится от указателей, но накладывает существенные ограничения на размер таблицы.

Каждая ячейка таблицы содержит либо ключ, либо значение NULL. В случае закрытого хэширования удаление элементов вызывает сложности, поэтому не стоит пользоваться этим способом. Также недостатком закрытого хэширования является возможность неудачного подбора хэш-функции, так что для вставки очередного элемента не найдется свободной ячейки.

Для вставки или поиска последовательно исследуются ячейки до тех пор, пока не встретится пустая ячейка.

Хэш-функция имеет зависит от двух параметров: $h'(x, i)$.

Алгоритм 3: Создание хэш-таблицы

Вход: A — массив размерности N, M — размерность хэш-таблицы
Выход: Хэш-таблица `hash`
начало алгоритма
- Создаем хэш-таблицу и заполняем ее значением `INF`;
- **цикл пока** *не дошли до конца массива* **выполнять**
  - Определяем значение вспомогательной хэш-функции $k = h(A[i])$;
  - `j = 0`;
  - **цикл пока** *пока не дошли до хэш-таблицы* **выполнять**
    - Определяем значение хэш-функции $p = f(k, j)$;
    - **если** *p ячейка хэш-таблицы не занята* **то**
      - Вставляем $A[i]$ в p-ую ячейку хэш-таблицы;
      - Прекращаем цикл;
    - **иначе**
      - Увеличиваем j;

конец алгоритма

Для поиска элемента определяем значение вспомогательной хэш-функции для этого элемента. И идем по соответствующим ячейкам таблицы до тех пор, пока не встретим искомый элемент или `NULL`.

1.2.1 Линейное хэширование

Пусть есть любая вспомогательная хэш-функция $h'(x)$, рассмотренная в предыдущей главе. Тогда будем рассматривать хэш-функцию вида: $h(x, i) = (h'(x) + i) \mod M$, где i принимает значения в диапазоне $[0, \ldots, M-1]$, M — размер хэш-таблицы ($M \geqslant N$).

Первой возможной ячейкой является та, которую дает вспомогательная хэш-функция, далее последовательно исследуются все ячейки, пока не встретится пустая. Возможно создание длинной последовательности занятых ячеек, ячеек, что удлиняет время поиска.

Например, пусть вспомогательной функцией является $h'(x) = x \mod M$. и $M = 20$. Рассмотрим пример из предыдущей главы:

Пример 1.3. Дан набор чисел: 12, 17, 25, 41, 23, 11, 24, 21, 26, 44, 33, 10, 20, 19, 29. Построить хэш-таблицу.

$N = 15$.

12: $h'(12) = 12$. Ячейка с индексом 12 пустая, можно заполнять.

17: $h'(17) = 17$. Ячейка с индексом 17 пустая, можно заполнять.

25: $h'(25) = 5$. Ячейка с индексом 5 пустая, можно заполнять.

41: $h'(41) = 1$. Ячейка с индексом 1 пустая, можно заполнять.

23: $h'(23) = 3$. Ячейка с индексом 3 пустая, можно заполнять.

11: $h'(11) = 11$. Ячейка с индексом 11 пустая, можно заполнять.

24: $h'(24) = 4$. Ячейка с индексом 4 пустая, можно заполнять.

21: $h'(21) = 1$. Ячейка с индексом 1 занята, увеличиваем индекс. Ячейка с индексом 2 пустая, можно заполнять.

26: $h'(26) = 6$. Ячейка с индексом 6 пустая, можно заполнять.

44: $h'(44) = 4$. Ячейка с индексом 4 занята, увеличиваем индекс. Ячейка с индексом 5 занята, увеличиваем индекс. Ячейка с индексом 6 занята, увеличиваем индекс. Ячейка с индексом 7 пустая, можно заполнять.

33: $h'(33) = 13$. Ячейка с индексом 13 пустая, можно заполнять.

10: $h'(10) = 10$. Ячейка с индексом 10 пустая, можно заполнять.

20: $h'(20) = 0$. Ячейка с индексом 0 пустая, можно заполнять.

19: $h'(19) = 19$. Ячейка с индексом 19 пустая, можно заполнять.

29: $h'(29) = 9$. Ячейка с индексом 9 пустая, можно заполнять.

Итого хэш-таблица имеет следующий вид:

0	20
1	41
2	21
3	23
4	24
5	25
6	26
7	44
8	NULL
9	29
10	10
11	11
12	12
13	33
14	NULL
15	NULL
16	NULL
17	17
18	NULL
19	19

□

1.2.2 Квадратичное хэширование

Выбираем хэш-функцию вида: $h(x, i) = (h'(x) + c_1 i + c_2 i^2) \mod M$.

Каждая следующая ячейка смещена относительно нулевой ячейки (значение $h'(x)$) на величину, характеризующуюся квадратичной зависимостью, что лучше линейной. На при неудачном выборе параметров c_1, c_2, m, может возникнуть ситуация, когда для элемента не окажется свободной ячейки, удовлетворяющей заданной хэш-функции.

Например, пусть вспомогательной функцией является $h'(x) = x \mod M$. и $M = 20$. Рассмотрим пример из предыдущей главы:

Пример 1.4. Дан набор чисел: 12, 17, 25, 41, 23, 11, 24, 21, 26, 44, 33, 10, 20, 19, 29. Построить хэш-таблицу.

$N = 15$. Пусть $M = 20$, $c_1 = 1$, $c_2 = 3$.

12: $h'(12) = 12$. Ячейка с индексом 12 пустая, можно заполнять.

17: $h'(17) = 17$. Ячейка с индексом 17 пустая, можно заполнять.

25: $h'(25) = 5$. Ячейка с индексом 5 пустая, можно заполнять.

41: $h'(41) = 1$. Ячейка с индексом 1 пустая, можно заполнять.

23: $h'(23) = 3$. Ячейка с индексом 3 пустая, можно заполнять.

11: $h'(11) = 11$. Ячейка с индексом 11 пустая, можно заполнять.

24: $h'(24) = 4$. Ячейка с индексом 4 пустая, можно заполнять.

21: $h'(21) = 1$. Ячейка с индексом 1 занята, увеличиваем индекс: $1 + 1 * 1 + 3 * 1 = 5$. Ячейка с индексом 1 занята, увеличиваем индекс: $1 + 1 * 2 + 3 * 4 = 15$. Ячейка с индексом 15 пустая, можно заполнять.

26: $h'(26) = 6$. Ячейка с индексом 6 пустая, можно заполнять.

44: $h'(44) = 4$. Ячейка с индексом 4 занята, увеличиваем индекс:$4 + 1 * 1 + 3 * 1 = 8$. Ячейка с индексом 8 пустая, можно заполнять.

33: $h'(33) = 13$. Ячейка с индексом 13 пустая, можно заполнять.

10: $h'(10) = 10$. Ячейка с индексом 10 пустая, можно заполнять.

20: $h'(20) = 0$. Ячейка с индексом 0 пустая, можно заполнять.

19: $h'(19) = 19$. Ячейка с индексом 19 пустая, можно заполнять.

29: $h'(29) = 9$. Ячейка с индексом 9 пустая, можно заполнять.

Итого хэш-таблица имеет следующий вид:

0	20
1	41
2	NULL
3	23
4	24
5	25
6	26
7	NULL
8	44
9	29
10	10
11	11
12	12
13	33
14	NULL
15	21
16	NULL
17	17
18	NULL
19	19

□

1.2.3 Двойное хэширование

В качестве хэш-функции выбираем функцию вида: $h(x, i) = (h_1(x) + ih_2(x)) \mod M$, где h_1 и h_2 — вспомогательные хэш-функции.

Начальная ячейка — это значение $h_1(x)$, а смещение — значение $h_2(x)$.

Для того, чтобы хэш-функция могла охватить всю таблицу, значение h_2 должно быть взаимно простым с размером хэш-таблицы. Вариантов выбора несколько: либо выбрать M степенью двойки, а h_2 сконструировать таким образом, чтобы она возвращала только нечетные значения; либо выбрать $h_1(x) = x \mod M$, а $h_2(x) = 1 + (x \mod M')$, где M — простое число, а $M' = M - 1$.

Данная функция реально зависит от двух параметров, поэтому является достаточно хорошей и содержит малое число цепочек занятых ячеек.

Рассмотрим пример из предыдущей главы, в качестве M выбираем простое число, например, 19:

Пример 1.5. Дан набор чисел: 12, 17, 25, 41, 23, 11, 24, 21, 26, 44, 33, 10, 20, 19, 29. Построить хэш-таблицу.

$N = 15$. Пусть $M = 19$, $M' = 18$.

12: $h_1(12) = 12$. Ячейка с индексом 12 пустая, можно заполнять.

17: $h_1(17) = 17$. Ячейка с индексом 17 пустая, можно заполнять.
25: $h_1(25) = 6$. Ячейка с индексом 6 пустая, можно заполнять.
41: $h_1(41) = 3$. Ячейка с индексом 3 пустая, можно заполнять.
23: $h'(23) = 4$. Ячейка с индексом 4 пустая, можно заполнять.
11: $h'(11) = 11$. Ячейка с индексом 11 пустая, можно заполнять.
24: $h'(24) = 5$. Ячейка с индексом 5 пустая, можно заполнять.
21: $h'(21) = 2$. Ячейка с индексом 2 пустая, можно заполнять.
26: $h'(26) = 7$. Ячейка с индексом 7 пустая, можно заполнять.
44: $h'(44) = 6$. Ячейка с индексом 6 занята, увеличиваем индекс:$h_2(44) = 9 \to h(44) = 15$. Ячейка с индексом 15 пустая, можно заполнять.
33: $h'(33) = 14$. Ячейка с индексом 14 пустая, можно заполнять.
10: $h'(10) = 10$. Ячейка с индексом 10 пустая, можно заполнять.
20: $h'(20) = 1$. Ячейка с индексом 1 пустая, можно заполнять.
19: $h'(19) = 0$. Ячейка с индексом 0 пустая, можно заполнять.
29: $h'(29) = 10$. Ячейка с индексом 10 занята, увеличиваем индекс:$h_2(29) = 12 \to h(29) = (10 + 12) \mod 19 = 3$. Ячейка с индексом 3 занята, увеличиваем индекс:$h(29) = (10 + 12 * 2) \mod 19 = 15$. Ячейка с индексом 15 занята, увеличиваем индекс:$h(29) = (10 + 12 * 3) \mod 19 = 8$. Ячейка с индексом 8 пустая, можно заполнять.

Итого хэш-таблица имеет следующий вид:

0	19
1	20
2	21
3	41
4	23
5	24
6	25
7	26
8	29
9	NULL
10	10
11	11
12	12
13	NULL
14	33
15	44
16	NULL
17	17
18	NULL

□