ОСНОВЫ ТЕОРИИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

Пышкин Евгений Валерьевич
к.т.н., доцент

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Блок 6. Обобщенные типы, шаблоны классов

Типичные операции с контейнерами

- Доступ (чтение/запись) к элементу по его номеру (индексу)
- Добавление нового элемента в конец контейнера, начало контейнера, вставка нового элемента в середину
- Удаление элемента из конца контейнера, из начала контейнера, из середины контейнера
- Поиск элемента по его значению

Массив (низкоуровневый)

- **int** arr[20]; // или
- **int*** arr = **new int**[size];
- Достоинства
  - высокая скорость доступа к отдельным элементам по индексу
  - высокая скорость вставки/удаления в конец
- Недостатки
  - ограниченный размер
  - низкая скорость вставки/удаления в начало и середину

Массив (высокоуровневый)


```
class Array
{
  // Увеличение размера
  void increaseSize(int newSize);
public:
  // ...
  // Индексация
  int operator[](int index) const;
  int& operator[](int index);
  // Вставка элемента
  void insert(int elem, int index);
  // Вставка в конец
  void insert(int elem);
  // Удаление элемента
  void remove(int index);
  // ...
};
```

Массив (высокоуровневый)

- Дополнительные возможности
  - изменение размера по необходимости
- Достоинства
  - скорость доступа к элементам по индексу остается высокой
  - скорость вставки/удаления в конец массива остается высокой
- Недостатки
  - скорость вставки/удаление в начало и середину массива остается низкой
  - скорость поиска остается низкой
- Как бороться с недостатками?

Примеры других контейнеров

- Очередь (двухсторонняя, односторонняя)
- Стек
- Дерево (бинарное, общее)
- Хэш-таблица
- Линейный список (однонаправленный, двунаправленный, кольцевой)

Очередь и стек

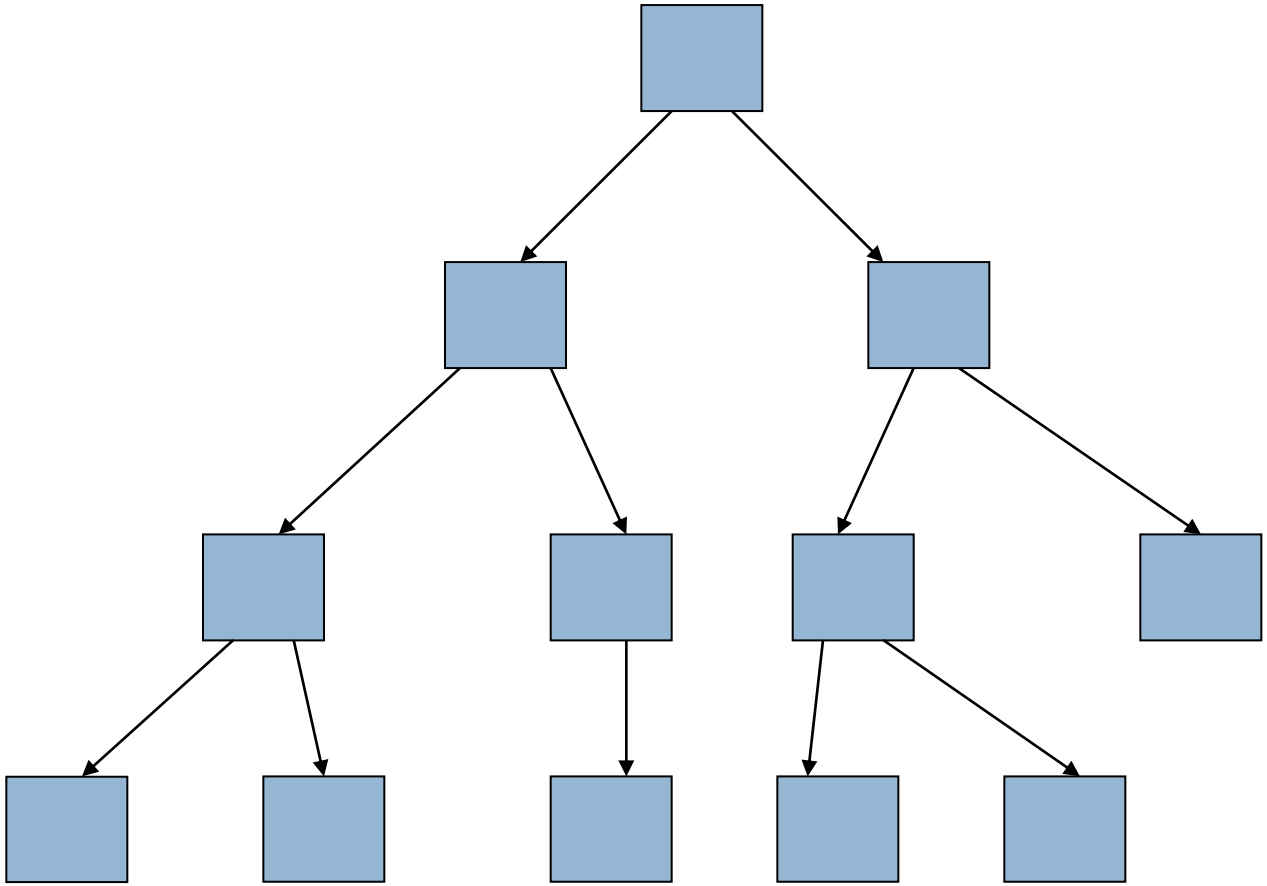
- Стек

- Односторонняя очередь (queue)

- Двусторонняя очередь (deque)

- Реализация?

Бинарное дерево



Хэш-таблица

ключ	данные
ключ	данные
ключ	данные

ключ	данные

- Если данные равны, то ключи равны
- Если данные различаются, то ключи, как правило, тоже различаются (за редкими исключениями)
- Хэш-функция - порождает ключ по данным
- Сравнение ключей существенно проще сравнения данных

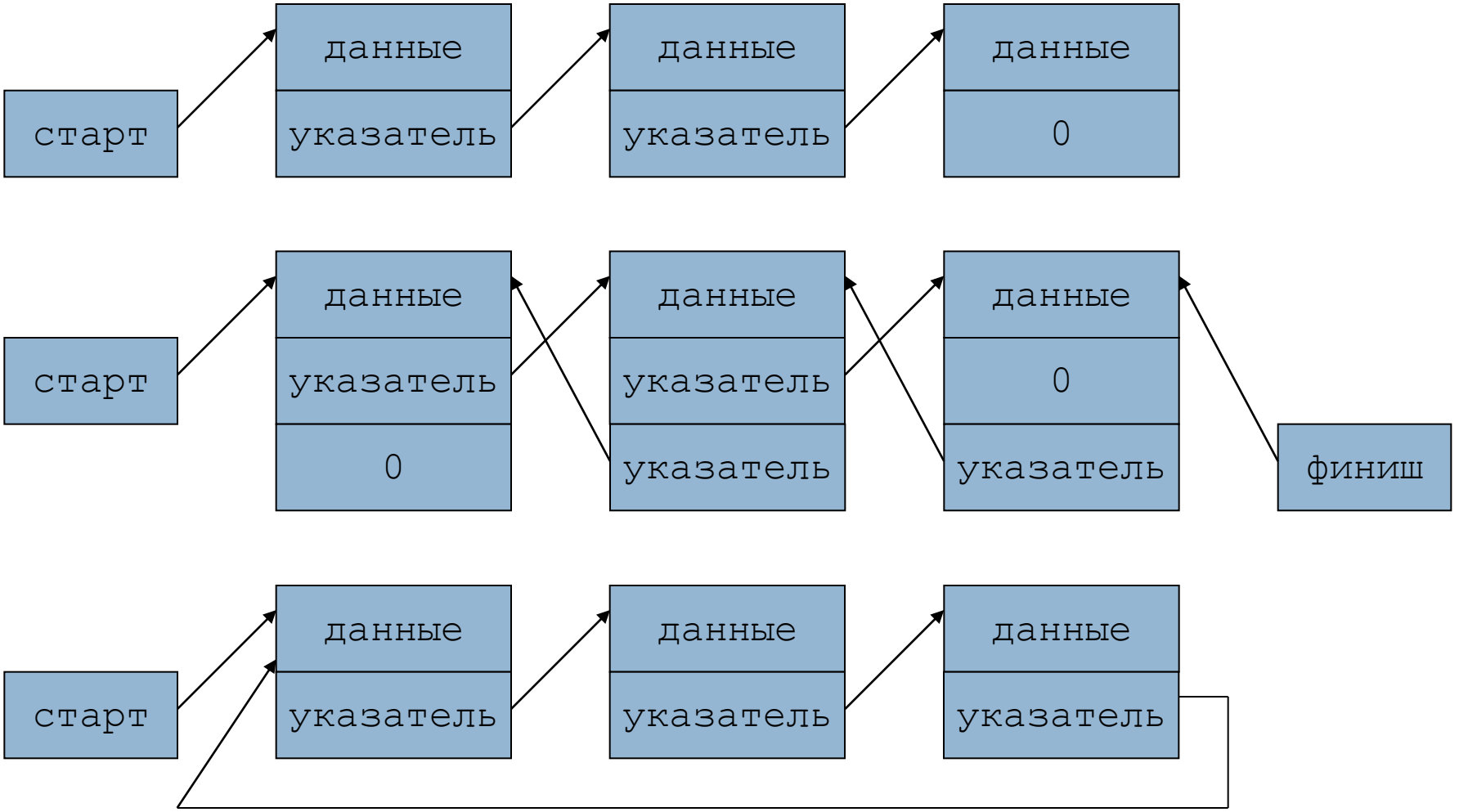
Принцип организации линейного списка

- Элементы списка могут располагаться в памяти не подряд
- Место под элементы выделяется и освобождается по необходимости
- Каждый элемент, помимо полезных данных, хранит указатель на соседний элемент
- Достоинства, недостатки?

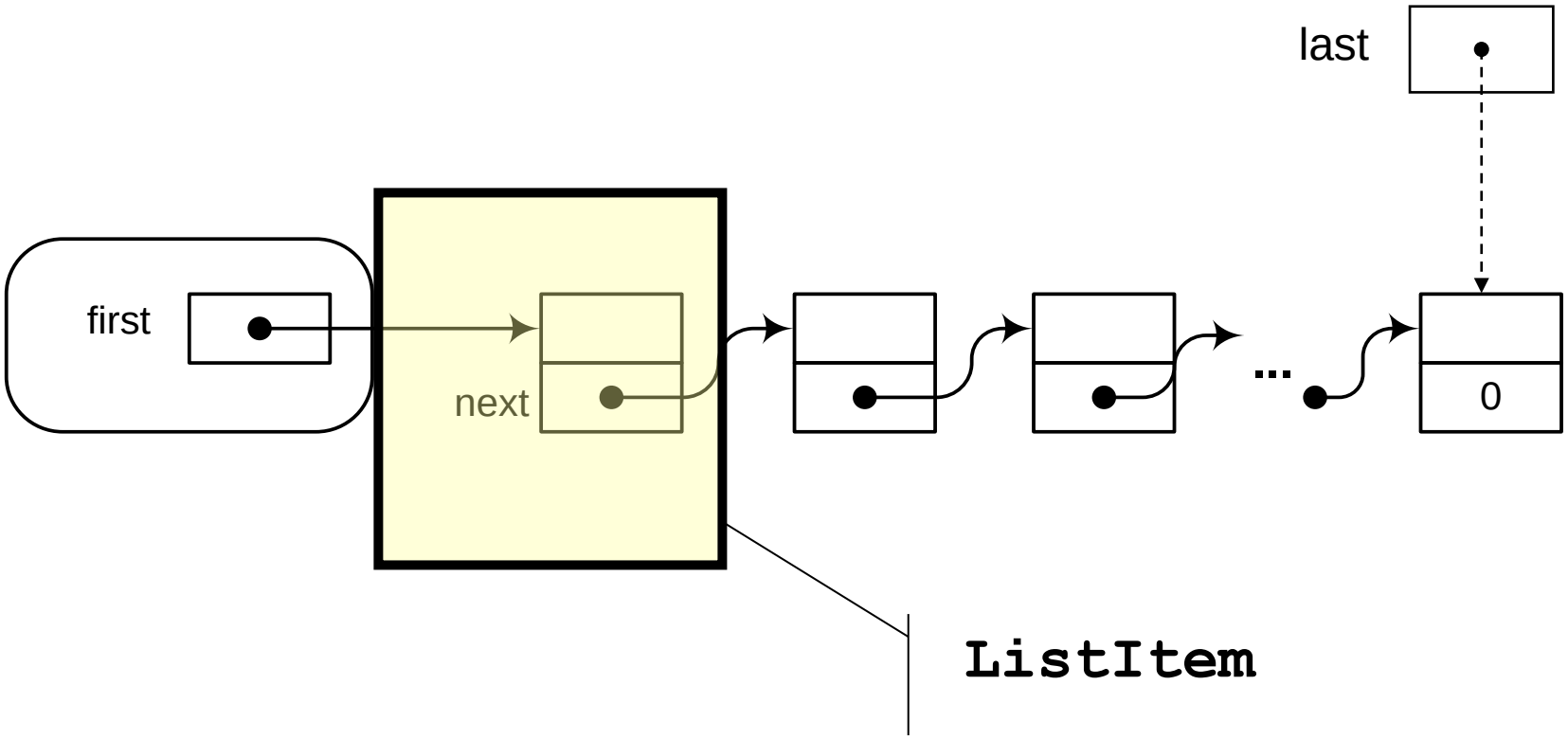
Линейный список - достоинства и недостатки

- Достоинства
  - высокая скорость операций вставки и удаления
  - возможность создавать списки любого размера
- Недостатки
  - низкая скорость операции доступа по индексу
  - низкая скорость поиска элементов

Виды линейных списков



Линейный однонаправленные список: общее представление



Линейный однонаправленный список: элемент списка

```
template <class ITEM>
struct ListItem
{
        ITEM data;
        ListItem<ITEM> *next;

        ListItem( const ITEM& data,
             ListItem<ITEM>* next = 0 )
        {
            this->data = data;
            this->next = next;
        }
};
```

Линейный однонаправленный список: определение класса

```
template <class T> class List
{
public:
        class RemoveItemException{};
        class InsertItemException{};
private:
        template <class ITEM>
        struct ListItem
        {
            // ...
        };
        ListItem<T> *first;
// ...
};
```

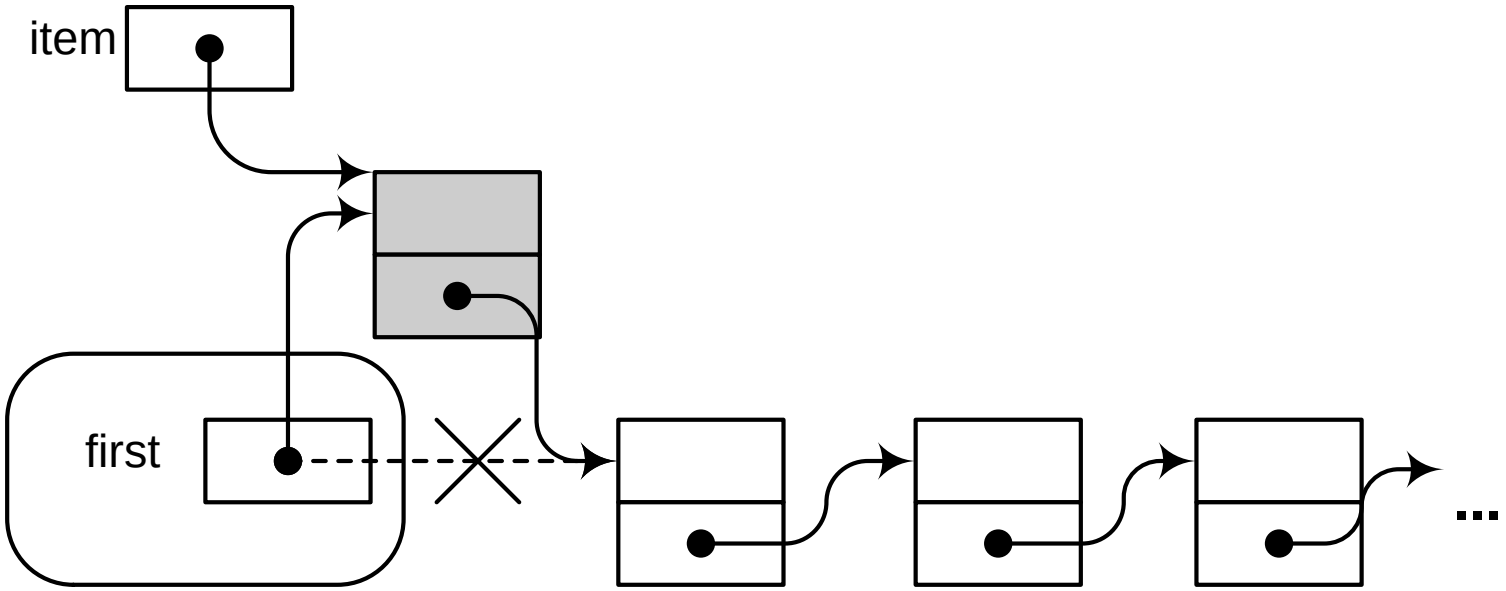
Инициализация

```
template <class T>
List::List()
{
        first = 0;

        // Возможно, еще какие-то действия
        // в зависимости от
        // содержания полей класса List
        // ...
}
```

Проверка наличия элементов

```
template <class T>
bool List::isEmpty()
{
        return first == 0;
}
```

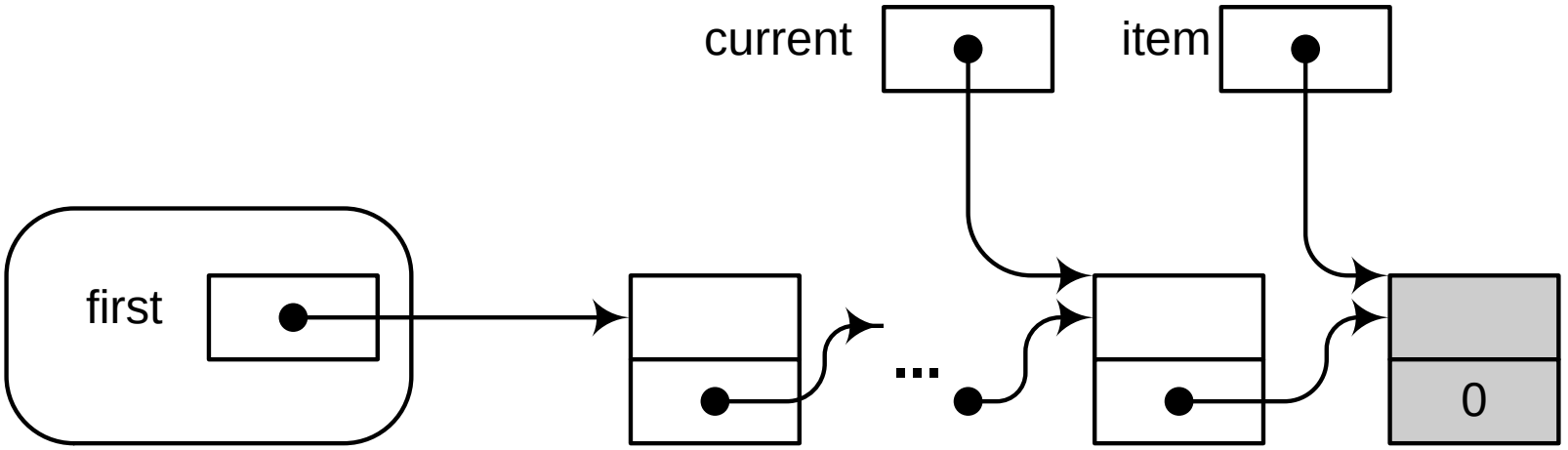
Добавление элемента в начало списка



Добавление элемента в начало списка

```
template <class T>
void List::pushBegin( const T& data )
{
        ListItem<T> *item =
            new ListItem<T>( data, first );

        first = item;
}
```

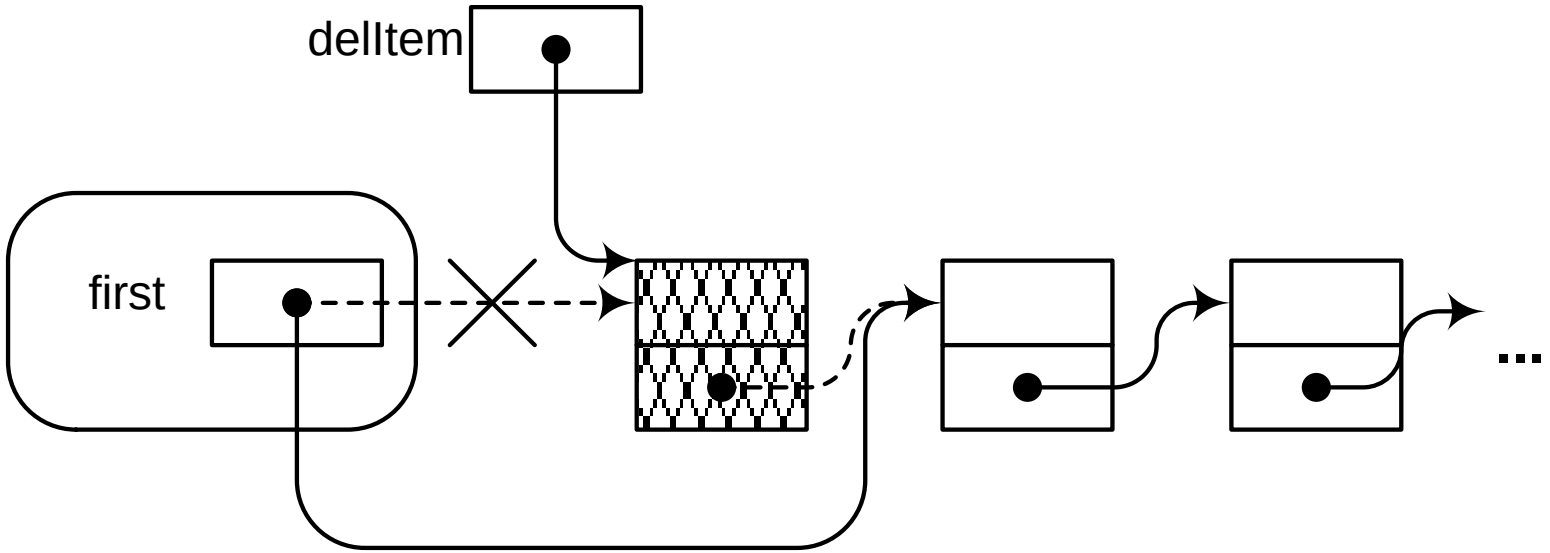
Добавление элемента в конец списка



Добавление элемента в конец списка

```
template <class T>
void pushEnd( const T& data )
{
        ListItem<T> *item = new ListItem<T>(data);

        if( first == 0 )
        {
            first = item;
            return;
        }

        ListItem<T> *current = first;
        while( current->next != 0 ) current = current->next;

        current->next = item;
        return;
}
```

Удаление элемента из начала списка



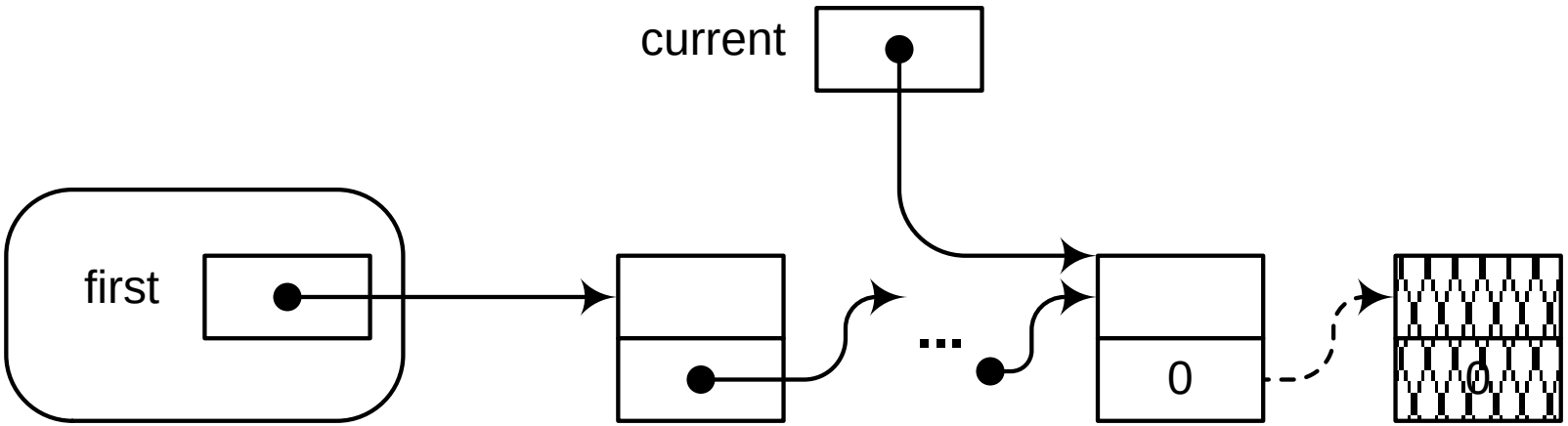
Удаление элемента из начала списка

```
template <class T>
void List::removeBegin()
{
        ListItem<T> *delItem;

        if( first == 0 )
            throw RemoveItemException();

        delItem = first;
        first = delItem->next;

        delete delItem;
        return;
}
```

Удаление всего списка

```
template <class T>
void List::removeAll()
{
        if( first == 0 )
            throw RemoveItemException();

        while( first != 0 )
        {
            removeBegin();
        }
       return;
}
```

Удаление элемента из конца списка



Удаление элемента из конца списка

```
template <class T>
void List::removeEnd()
{
        ListItem<T> *current;

        if( first == 0 )
            throw RemoveItemException();

        current = first;
        if( current->next == 0 )
            return removeBegin();

        // Продолжение на следующем слайде...
}
```

Удаление элемента из конца списка

```
/* template <class T> void List::removeEnd()
   // Продолжение
*/
      // Цикл поиска элемента,
      // предшествующего последнему
      while( current->next->next != 0 )
            current = current->next;

      // Собственно удаление последнего элемента
      delete current->next;
      current->next = 0;

      return;
}
```

Поиск элемента (есть или нет)

```
template <class T>
bool List::hasItem( const T& dataToFind )
{
        ListItem<T> *current;

        current = first;
        while( current != 0 )
        {
              if( current->data == dataToFind )
                    return true;
             current = current->next;
        }
        return false;
}
```

Реализация класса: демонстрационный пример и анализ

- Проектные проблемы
  - Невозможно реализовать в классе List все операции над элементами списка, которые только могут потребоваться
  - Содержание операции над объектом данных, хранящихся в списке, зависит от структуры этого объекта
  - Необходимо дать возможность работать с отдельными объектами данных, хранящихся в списке, но не с внутренней реализацией списка
- Такие проблемы характерны и для других вмещающих типов

- Вернемся к этим проблемам позднее

Достоинства и недостатки однонаправленного списка

- Достоинства
  - быстро вставляет и удаляет **после**
  - быстро вставляет и удаляет **в начале**
- Недостатки
  - медленно вставляет и удаляет **перед**
  - медленно вставляет и удаляет **в конце**
- Как справиться с недостатками?

Последовательная обработка содержимого контейнера

- Часто необходимо просмотреть все элементы контейнера
- Для этой цели определяют итераторы

Итерация списка

```
template <class T>
class ListIt {
    // ...
public:
    class BadIteratorException{};
    void start() {
        currentIt=first;
    }
    bool hasMore() {
        return currentIt!=0;
    }
    void next() {
        if( currentIt == 0 ) throw BadIteratorException();
        currentIt = currentIt->next;
    }
    T* get() {
        if( currentIt == 0 ) throw BadIteratorException();
        return &(currentIt->data);
private:
    ListItem<T> *first;
    ListItem<T> *currentIt;
};
```

Пример использования: внешняя реализация HasItem

```
template <class T>
bool HasItem( ListIt<T>& list, const T& checkData )
{
    for( list.start(); list.hasMore(); list.next() )
        // Необходима точная согласованность вызовов
    {
        T* ptrItem = list.get();
        if( *ptrItem == checkData ) return true;
    }
    return false;

    // В каждый момент времени над списком может быть
    // запущен только ОДИН итератор – это недостаток,
    // присущий внутреннему итератору
}
```

Построение внешнего итератора

- Информация о текущем положении итератора хранится в самостоятельном объекте
- Поскольку итератор использует внутреннее представление списка, обычно класс итератора определяют внутри класса списка (с предоставлением итератору дружественного доступа)
- Можно определить несколько классов итераторов, реализующих различные варианты итерации
  - Прямая итерация
  - Обратная итерация
  - Итерация вставки
  - …
- Такая модель используется в стандартной библиотеке

Стандартная библиотека шаблонов

- Одной из важнейших составляющих языка C++ является стандартная библиотека шаблонов (standard template library, STL)
- STL содержит большое количество готовых шаблонов, которые могут быть использованы при написании программ

Использование готовых шаблонов из библиотеки STL

- Все шаблоны STL определены в пространстве имен std
  - **using namespace** std;
- Как правило, использование шаблона требует подключения одноименного заголовочного файла
  - **#include** <vector>
  - **#include** <list>

Готовые шаблоны контейнеров

- Основные
  - vector - массив переменного размера
  - list - список
  - deque - двухсторонняя очередь
- Дополнительные (адаптеры контейнеров)
  - queue - односторонняя очередь
  - stack - стек
- Специальные
  - basic_string - строка
- Ассоциативные
  - map - таблица ключ-значение
  - set - множество

Требования к элементам контейнера

- открытый конструктор копирования
- открытый оператор присваивания
- открытый деструктор
- отдельные операции также требуют
  - открытый конструктор по умолчанию
  - открытый оператор сравнения на равенство
  - открытый оператор сравнения <

Шаблон <vector>

- Внутренняя организация - на основе массива
- Размер массива увеличивается по мере необходимости

Основные свойства вектора

- size() - *размер* - количество элементов, имеющихся в векторе на данный момент
  - может быть изменено вызовом resize(int)
  - оператор индексации позволяет обратиться к элементам 0…size()-1
- capacity() - *вместимость* - количество элементов, под которые на данный момент выделена память
  - всегда верно capacity() >= size()
  - может быть изменено вызовом reserve(int)
- max_size() - максимально возможное количество элементов в векторе

Основные возможности вектора

- Конструкторы
  - по умолчанию (пустой вектор)
  - заданного размера
  - заданного размера и наполнения
  - копирования
- Оператор присваивания
- Операторы сравнения
- Обращение по индексу
  - диапазон значений 0…size()-1
  - при неверном индексе происходит ошибка при выполнении программы

Основные возможности вектора

- Очистка (clear)
- Проверка на пустоту (empty)
- Добавление последнего элемента (push_back)
  - размер увеличивается на 1, вместимость увеличивается при необходимости
- Удаление последнего элемента (pop_back)
  - размер уменьшается на 1

Пример использования вектора - чтение файла с целыми числами


```
vector<int>* readValues(const char* fname)
{
  ifstream in(fname);
  if (!in.is_open())
    return 0;
  vector<int>* arr = new vector<int>();
  int val;
  in>>val;
  while (!in.fail())
  {
    arr->push_back(val);
    in>>val;
  }
  return arr;
}
```

Пример использования вектора - сортировка


```
void orderValues(vector<int>& arr)
{
  for (size_t last=arr.size()-1; last>=1; last--) {
    size_t maxindex=0;
    int maxvalue=arr[0];
    for (size_t i=1; i<=last; i++) {
      if (arr[i]>maxvalue) {
        maxvalue=arr[i];
        maxindex=i;
      }
    }
    arr[maxindex]=arr[last];
    arr[last]=maxvalue;
  }
}
```

Достоинства и недостатки вектора

- Достоинства
  - Быстрый доступ по индексу
  - Быстрые операции с последним элементом (вставка, удаление)
- Недостатки
  - Медленные операции с первым и внутренними элементами (вставка, удаление)

Шаблон <list>

- Внутренняя организация - на основе двунаправленного линейного списка
- Элементы добавляются по мере необходимости

Основные свойства списка

- size() - количество элементов в списке на данный момент
  - может быть изменено вызовом resize()
- max_size() - максимально возможное количество элементов в списке

Основные возможности списка

- Конструкторы
  - по умолчанию (пустой список)
  - заданного размера
  - заданного размера и наполнения
  - копирования
- Оператор присваивания
- Операторы сравнения
- *Обращение по индексу*
  - для списка данная функция **не поддерживается**

Основные возможности списка

- Очистка (clear)
- Проверка на пустоту (empty)
- Добавление последнего элемента (push_back) или первого элемента (push_front)
  - размер увеличивается на 1
- Удаление последнего элемента (pop_back) или первого элемента (pop_front)
  - размер уменьшается на 1

Основные возможности

- изменение порядка элементов на обратный (reverse)
- сортировка (sort)
- слияние двух списков с упорядочением (merge)
- удаление дублирующихся элементов (unique)

Итераторы

- Специальные объекты, предназначенные для перебора элементов контейнера; по принципу действия напоминают указатели
- Для списка - фактически, единственное средство доступа
- Определяются как:
  - list<int>::iterator it; // *или*
  - vector<float>::iterator it2;

Возможности итераторов

- Разыменование (*it)
- Перевод на следующий элемент (it++)
- Перевод на предыдущий элемент (it--)
- Сложение, вычитание
  (it+2, it-3, it2-it1,
  только для вектора)

Перебор элементов списка с помощью итератора


```
list<int> mylist;
// ...
int sum = 0;
for (list<int>::iterator it=mylist.begin();
                         it!=mylist.end(); it++)
{
  cout<<*it<<' ';
  sum += *it;
}
cout<<endl<<sum<<endl;
```

Возможности вектора и списка, связанные с итераторами

- begin() - получение итератора, указывающего на 1-й элемент
- end() - получение итератора, указывающего на «воображаемый» элемент, следующий **за** последним
- insert(it, value) - вставка **перед** it элемента value
- insert(it, itfrom, itto) - вставка **перед** it элементов [itfrom, itto)
- assign(itfrom, itto) - очистка с последующим заполнением элементами [itfrom, itto)

Достоинства и недостатки списка

- Достоинства
  - Быстрые операции со случайными элементами (вставка, удаление)
- Недостатки
  - Отсутствие доступа по индексу
  - Перебор элементов происходит медленно

Шаблон <deque>

- Внутренняя организация - на основе кольцевого массива или нескольких кольцевых массивов

Основные свойства очереди

- size() - количество элементов в списке на данный момент
  - может быть изменено вызовом resize()
- max_size() - максимально возможное количество элементов в списке

Основные возможности очереди

- Конструкторы
  - по умолчанию (пустая очередь)
  - заданного размера
  - заданного размера и наполнения
  - копирования
- Оператор присваивания
- Операторы сравнения
- Обращение по индексу
  - диапазон значений 0…size()-1
  - при неверном индексе происходит ошибка при выполнении программы

Основные возможности очереди

- Очистка (clear)
- Проверка на пустоту (empty)
- Добавление последнего элемента (push_back) или первого элемента (push_front)
  - размер увеличивается на 1
- Удаление последнего элемента (pop_back) или первого элемента (pop_front)
  - размер уменьшается на 1
- Основные операции с итераторами (begin, end, insert, assign)

Достоинства и недостатки очереди

- Достоинства
  - Быстрые операции с первым и последним элементом (вставка, удаление)
  - Быстрый доступ по индексу (хотя и уступающий вектору)
- Недостатки
  - Медленные операции с внутренними элементами (вставка, удаление)

Вспомогательные контейнеры - stack, queue

- Реализуются на основе одного из основных контейнеров, обычно на основе *deque*
- Основные возможности
  - конструкторы
  - проверка на пустоту empty()
  - получение размера size()
  - добавление элемента push()
  - удаление элемента pop()
  - получение верхнего элемента top()

Строки STL

- Строка, состоящая из 8-битных символов (char), использующих одну из 8-битных кодировок, например, *CP-1251*
  - basic_string<char> *// или просто*
  - string
- Строка, состоящая из 16-битных символов (wchar_t), использующих кодировку *Unicode*
  - basic_string<wchar_t> *// или просто*
  - wstring
- Требуют подключения <string>

Основные возможности строк STL

- Конструкторы
  - по умолчанию
  - из низкоуровневой строки (char*)
  - копирования
- Операторы
  - присваивания =, assign
  - добавления (конкатенации) +=, append
  - индексации []
  - сравнения
- Получение размера - length, size
- Получение низкоуровневой строки - c_str()

Основные возможности строк STL

- Вставка строки insert(index, string)
- Получение подстроки substr(begin, length)
- Поиск подстроки find(string)
- Сравнение compare(string)
- Работа с итераторами

Пример работы со строками - сортировка файла по алфавиту

- Задача разбивается на три:
  - чтение строк из файла
  - сортировка строк
  - запись строк в файл
- Для хранения строк применим список - в этом случае функция сортировки будет готовой

Чтение строк из файла


```
list<string>* readFile(const char* filename)
{
  ifstream in(filename);
  if (!in.is_open())
    return 0;
  list<string>* result = new list<string>();
  const int MAX_LEN = 256;
  char str[MAX_LEN];
  in.getline(str, MAX_LEN);
  while (!in.fail())
  {
    result->push_back(str);
    in.getline(str, MAX_LEN);
  }
  return result;
}
```

Запись строк в файл


```
bool writeFile(list<string>* pList, const char* filename)
{
  ofstream out(filename);
  if (!out.is_open())
    return false;
  for (list<string>::iterator it=pList->begin();
      it!=pList->end(); it++)
  {
    out<<*it<<endl;
  }
  return true;
}
```

Главная функция


```
int main(void)
{
  list<string>* pList = readFile("in.txt");
  if (!pList) {
    cout<<"File in.txt does not exists"<<endl;
    return -1;
  }
  pList->sort();
  if (!writeFile(pList, "out.txt")) {
    cout<<"Cannot open output file"<<endl;
    return -2;
  }
  cout<<"Success!"<<endl;
  return 0;
}
```

STL и задачи поиска

- Имеется некоторое количество уникальных **ключей**; с каждым из ключей связаны некоторые **данные**. Требуется найти данные по заданному ключу
- Примеры
  - ФИО → номер телефона, адрес
  - адрес → владелец
  - …

Пример задачи

- Входной файл содержит имена, адреса и телефоны людей
- С клавиатуры вводится имя человека
- Необходимо найти его адрес и телефон
- Начнем с определения необходимых данных

Разделим ключ и данные


```
// Ключ будет просто string
// Данные
struct HumanData
{
  string address;
  string phone;
  HumanData();
  HumanData(string addr, string ph);
};

istream& operator >>(istream& in, string& str);
istream& operator >>(istream& in,
  HumanData& data);
ostream& operator <<(ostream& out,
  const HumanData& data);
```

Разделим ключ и данные


```
// Данные вместе с ключом
struct Human
{
  string name;
  HumanData data;
  Human();
  Human(string aname, string addr, string ph);
};

istream& operator >>(istream& in,
  Human& human);
ostream& operator <<(ostream& out,
  const Human& human);
```

Конструкторы

- Human::Human(
  string aname, string addr, string ph):
- name(aname), data(addr, ph) {}

- HumanData::HumanData(
  string addr, string ph)
- {
- address = addr;
- phone = ph;
- }

Функция ввода


```
istream& operator >>(istream& in, string& str)
{
  const int MAX_LEN=250;
  char buf[MAX_LEN];
  in.getline(buf, MAX_LEN);
  str = buf;
  return in;
}
istream& operator >>(istream& in, HumanData& data)
{
  in>>data.address>>data.phone;
  return in;
}
```

Чтение данных из файла


```
bool readHumans(
  const char* filename, vector<Human>& humans)
{
  ifstream in(filename);
  if (!in.is_open())
    return false;
  Human human;
  in>>human;
  while (!in.fail())
  {
    humans.push_back(human);
    in>>human;
  }
  return true;
}
```

Прямой поиск


```
class FindException {};
// Простой перебор всех вариантов
Human findByName(const vector<Human>& humans,
  const string& name)
{
  cout<<"Ищем: "<<name<<endl;
  for (vector<Human>::const_iterator
    it=humans.begin(); it!=humans.end(); it++)
  {
    if (it->name==name)
      return (*it);
  }
  throw FindException();
}
```

Демонстрация работы

- См. пример

Перевод OEM → ANSI


```
#undef UNICODE
#include <windows.h>

string convert(const char* str)
{
  char* buf = new char[strlen(str)+1];
  OemToChar(str, buf);
  string res(buf);
  delete[] buf;
  return res;
}
```

Результат

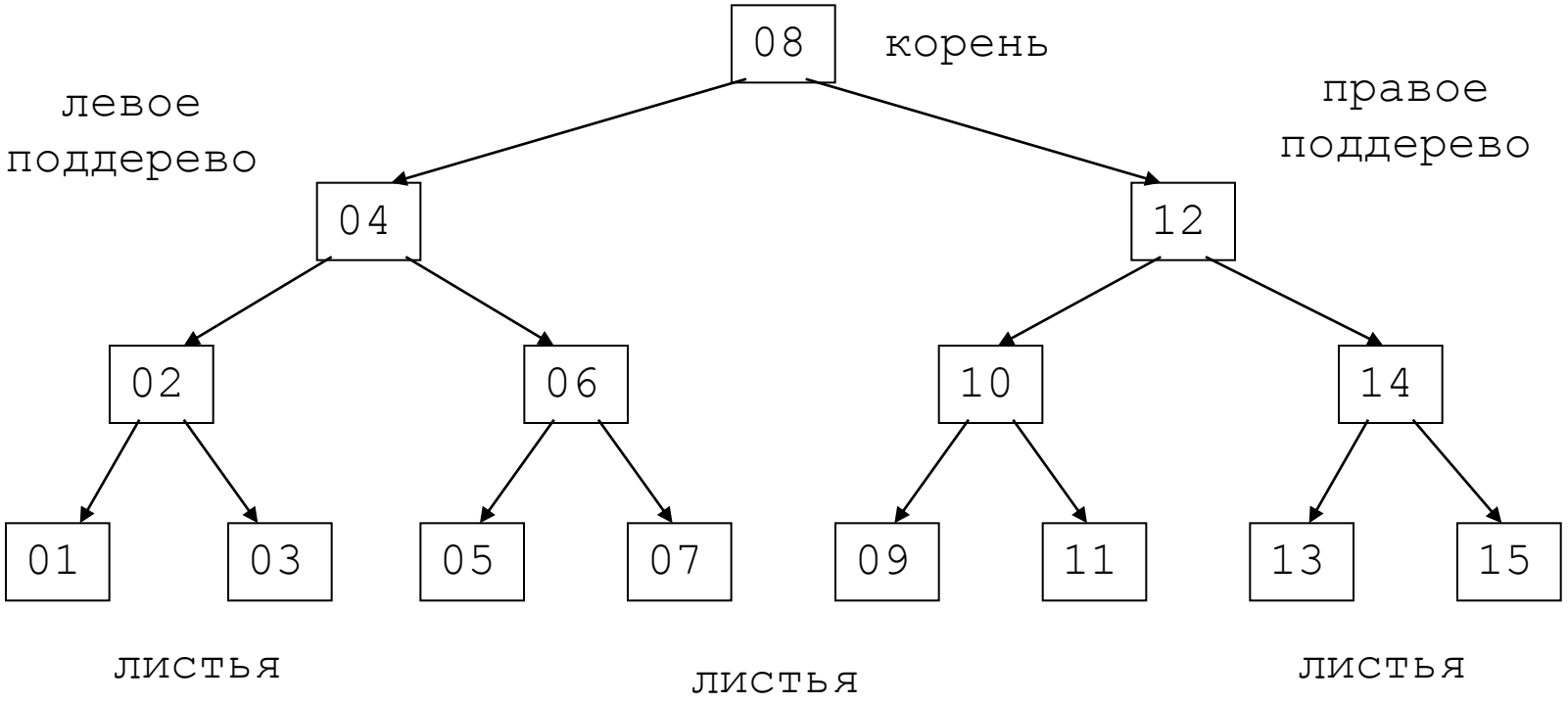
- Поиск осуществляется
- В среднем требуется N/2 сравнений, где N - размер контейнера

Сложные методы поиска

- Значительно ускоряют процесс поиска (то есть, требуют меньшее количество сравнений)
- Однако, все они требуют предварительной подготовки исходного массива

Бинарный поиск

- 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15



Бинарный поиск

- Идея - каждое сравнение уменьшает область поиска вдвое
- Результат - количество сравнений уменьшается до $\log_2 N$
- Требование - данные должны быть отсортированы по ключу

Реализация бинарного поиска


```
Human findByName(const vector<Human>& humans,
  const string& name, int min, int max)
{
  int index = (min+max)/2;
  if (min>max)
    throw FindException();
  else if (humans[index].name==name)
    return humans[index];
  else if (humans[index].name>name)
    return findByName(humans, name, min, index-1);
  else
    return findByName(humans, name, index+1, max);
}
```

Реализация бинарного поиска


```
Human findByName(const vector<Human>& humans,
  const string& name)
{
  cout<<"Ищем: "<<name<<endl;
  return findByName(humans,
    name, 0, (int)humans.size()-1);
}
```

Применимость

- Бинарный поиск актуален в том случае, когда данных большое количество
- Необходимо уже при первоначальном чтении данных создавать отсортированный список

Шаблон <map>

- Ассоциативный массив ключ-значение
  map<KEY, VALUE>
- Внутренняя организация - на основе линейного списка
- Автоматическая сортировка
- Реализация бинарного поиска
- Итераторы используют value_type
- **struct** value_type
- {
- KEY first;
- VALUE second;
- };

Основные методы

- size() - текущий размер
- map[key]=value - вставка
- it=find(key) - поиск ключа, возвращает итератор, указывающий на соответствующий элемент (или за последний элемент)
- begin() - итератор на 1-й элемент
- end() - итератор за последний элемент

Применение

- Пусть KEY будет string
- Пусть VALUE будет HumanData
- Тогда контейнер humans будет иметь тип map<string, HumanData>

Функция чтения


```
bool readHumans(const char* filename,
  map<string, HumanData>& humans)
{
  ifstream in(filename);
  if (!in.is_open())
    return false;
  string name;
  HumanData data;
  in>>name>>data;
  while (!in.fail())
  {
    humans[name] = data;
    in>>name>>data;
  }
  return true;
}
```

Поиск


```
char str[250];
cout<<"Введите фамилию, имя, отчество: "<<endl;
cin.getline(str, 250);
string name = convert(str);
map<string, HumanData>::const_iterator it =
  humans.find(name);
if (it!=humans.end())
{
  cout<<"Человек найден"<<endl;
  cout<<it->first<<' '<<it->second<<endl;
} else
{
  cout<<"Человек не найден: "<<name<<endl;
}
```

Проблемы бинарного поиска

- Не все ключи так легко упорядочить
- Для сложных ключей сравнение на равенство и неравенство может требовать много времени
- Поддержка отсортированного списка также может требовать времени

Хэш-поиск

- Идея состоит в замене сравнения ключей сравнением целых чисел
- С этой целью определяется hash(key) со следующими свойствами:
  - если key1==key2, то hash(key1)==hash(key2)
  - если hash(key1)==hash(key2),
    то, как правило, key1==key2

Организация хэш-поиска

- В простом варианте хэш хранится вместе с ключом и данными
- Перед тем, как сравнивать ключи, мы сравниваем хэши
  - если хэши разные, ключи уже можно не сравнивать
  - если хэши совпадают, то проверяем, совпадают ли ключи

Пример для нашей задачи


```
struct Human
{
  HumanData data;
  Human();
  Human(string aname, string addr, string ph);
  bool operator ==(const Human& human) const;
  string getName() const;
  void setName(const string& aname);
private:
  string name; // стало private - почему?
  int hash;
  void calcHash();
  friend istream& operator >>(istream& in,
    Human& human); // стал friend - почему?
};
```

Хэш-функция


```
void Human::calcHash()
{
  hash = 0;
  for (string::iterator it=name.begin();
        it!=name.end(); it++)
  {
    hash += (int)(*it);
  }
}
// В чем недостатки? Как лучше?
```

Функция сравнения


```
bool Human::operator ==(
  const Human& human) const
{
  return (hash==human.hash &&
          name==human.name);
}
```

Изменения name


```
Human::Human(string aname, string addr, string ph):
  name(aname), data(addr, ph)
{
  calcHash();
}
void Human::setName(const string& aname)
{
  name = aname;
  calcHash();
}
istream& operator >>(istream& in, Human& human)
{
  in>>human.name>>human.data;
  human.calcHash();
  return in;
}
```

Функция поиска


```
Human findByName(const vector<Human>& humans,
                 const string& name)
{
  cout<<"Ищем: "<<name<<endl;
  Human toFind(name, "", "");
  for (vector<Human>::const_iterator it=humans.begin();
       it!=humans.end(); it++)
  {
    if ((*it)==toFind)
      return (*it);
  }
  throw FindException();
}
```

Результат

- Число сравнений осталось таким же, как и в прямом поиске
- Однако, трудоемкость каждого сравнения уменьшилась
- Можно ли скомбинировать с бинарным поиском?

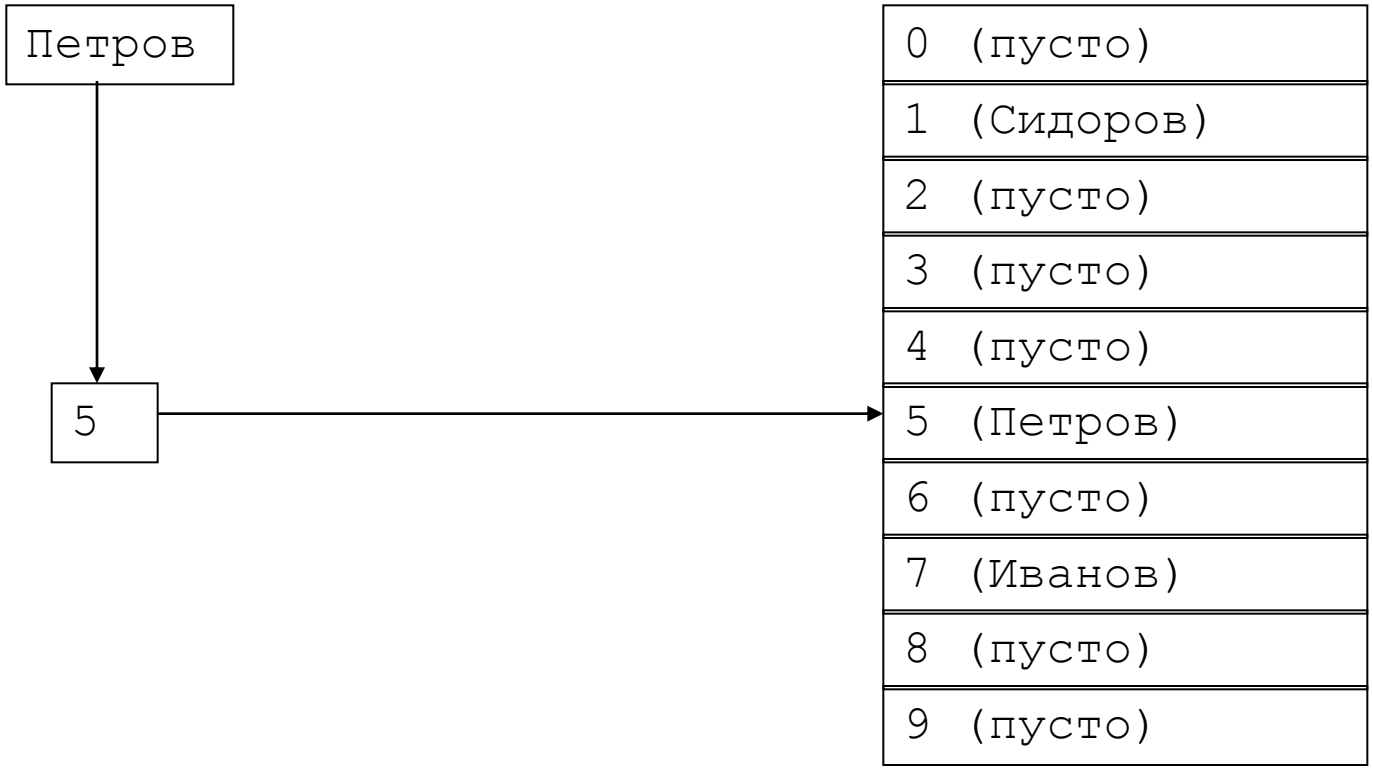
Развитие

- Ничто не мешает упорядочить наши объекты (вначале по значению хэша, а уже потом по значению ключа)
- После чего можно применить комбинацию бинарного и хэш-поиска

Идея сложного хэш-поиска

- Превратить результат хэш-функции в индекс массива
- Когда хотим что-либо найти, считаем хэш-функцию от ключа и обращаемся по этому индексу
- При идеальной хэш-функции мы все найдем **с первой же попытки**

Идея сложного хэш-поиска



Трудности сложного хэш-поиска

- Хэш-функцию необходимо ограничить по модулю
  - на практике обычно берут ее младшие биты
- Значения хэш-функции (тем более ограниченной) могут совпасть - **хэш-коллизия**

Разрешение хэш-коллизий

- Занять следующую свободную ячейку массива
- При поиске, проверяем значение ключа ячейки, полученной по хэш-индексу
- Если ключ не совпадает, последовательно проверяем следующие ячейки

Ступенчатая организация

Ступенчатая организация

- Если один из ящиков переполняется, у нас есть два решения:
  - увеличить вдвое число ящиков, при этом каждый из существующих разбивается на два в соответствии со значением хэш-функции
  - увеличить размер ящиков (неэффективно, но зато всегда работает)

Шаблон <hash_map>

- Нестандартный, требует подключения пространства имен stdext:
  - **#include** <hash_map>
  - **using namespace** stdext;
- Ассоциативный массив ключ-значение hash_map<KEY, VALUE>
- Ступенчатая организация хэш-таблицы
- Реализация хэш-поиска
- По набору методов похож на map