Ассоциативные массивы

Массив – контейнер данных, в котором элементы доступны по целочисленным индексам.

Ассоциативный массив – контейнер данных, в котором элементы доступны по индексам (ключам) имеющим произвольный тип

Множество – вырожденный случай ассоциативного массива, имеющий только ключи (без элементов)

Телефонный справочник

```
#include <iostream>
#include <string>
#include “phonebook.h”

using namespace std;

int main (){
        PhoneBook pBook;
        pBook.addRecord (“Vasya”, “121212”);
        pBook.addRecord (“Petya”, “212121”);
        cout << pBook.getRecord (“Vasya”);
        return 0;
}
```

Использование класса map

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main (){
        map <string, string> pBook;
        pBook [“Vasya”] = “121212”;
        pBook [“Petya”] = “212121”;
        cout << pBook [Vasya”];
        return 0;
}
```

Доступ к элементам ассоциативного массива

operator [] (const key_type&); - доступ к элементу по ключу. Если элемент с данным ключом отсутствует, то он создается и инициализируется конструктором по умолчанию

iterator find (const key_type&); - поиск элемента по ключу. Если ключ отсутствует, то возвращается Итератор, равный end ()

const_iterator find (const key_type&) const;

Итераторы ассоциативных массивов

Операции инкремента/декремента для итераторов не гарантируют никакого определенного порядка обхода.

Результатом разыменования итератора является пара ключ-значение:

```
template <typename first_type, typename second_type>
struct pair{
        first_type first;
        second_type second;
};
```

Вставка элементов в ассоциативный массив

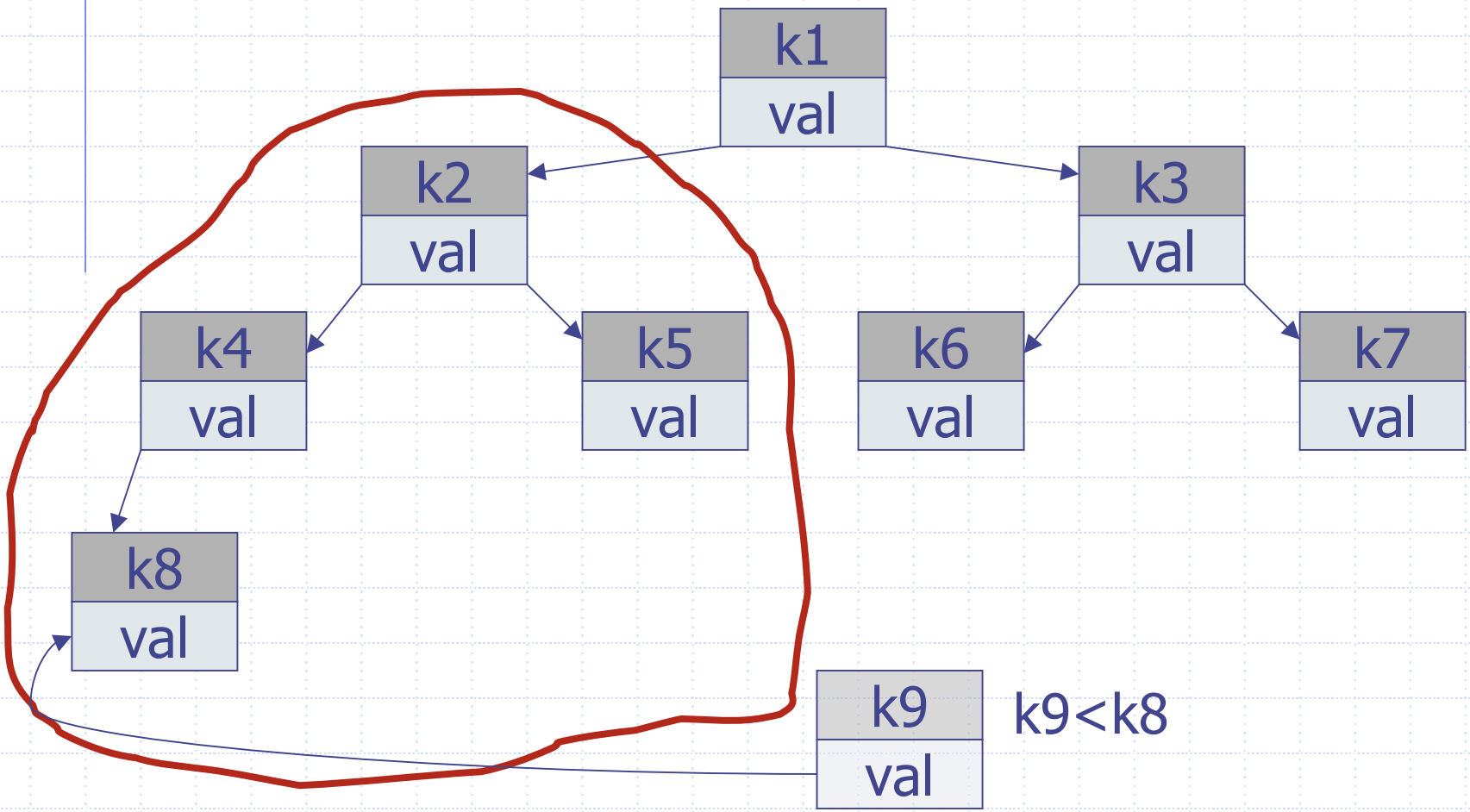
operator [] (const key_type&);

insert (const key_type&, const& value_type);

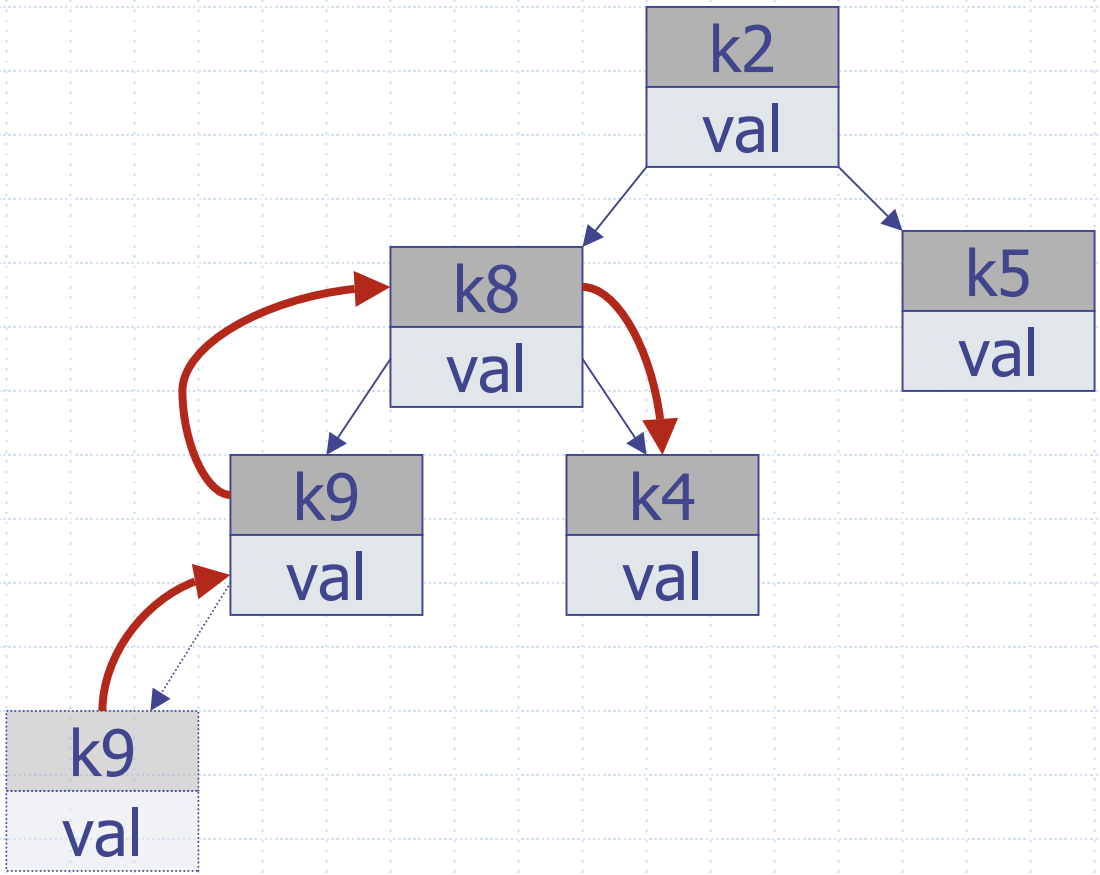
insert (const pair<key_type, value_type> &);

Реализация ассоциативного массива

Сбалансированное бинарное дерево



Балансировка дерева



Ассоциативные массивы: особенности реалиации

- Сложность операций доступа, добавление и удаления $O(\log_2 N)$

- Для ключей должна быть определена операция сравнения (operator <) таким образом, что если a < b – истина, то b < a – ложь, и если a < b < c, то a < c; а также сравнение на равенство (operator ==)

Множества

- Множество – это ассоциативный массив, у которого отсутствуют значения

```
template <typename key_type>
class set{…};
```

- Класс set обладает таким же функционалом, что и класс map за исключением следующего:
  - Отсутствует operator []
  - pair заменяется на key_type

Применение множеств

Множества – альтернатива массивам и спискам

Множества применяются, если одинаково часто используются операции вставки/удаления и доступа к элементам. При этом порядок элементов и их ключи не несут информации

multimap

- Каждому ключу может соответствовать несколько значений
- operator [] отсутствует
- Результатом любого обращения по ключу является набор значений.

Доступ к элементам multimap и multiset

```
iterator lower_bound (const key_type&);
//первый ключ с заданным ключом

iterator upper_bound (const key_type&);
//следующий элемент с ключом больше заданного

size_type count (const key_type&);
//количество элементов с заданным ключем

pair<iterator, iterator> equal_range (const key_type&);
```

Другие возможности C++

- Пространства имен
- Преобразование типов в стиле C++. RTTI
- Обработка ошибок. Исключения.

Пространства имен

```
//myclass.h
namespace Denis{
        class MyClass{
        public:
                MyClass ();
        };
}

Denis::MyClass::MyClass (){
        //…
}
```

```
#include “myclass.h”
int main (){
        Denis::MyClass a;
        return 0;}
```

```
#include “myclass.h”
using namespace Denis;
int main (){
        MyClass a;
        return 0;}
```

```
#include “myclass.h”
using Denis::MyClass;
int main (){
        MyClass a;
        return 0;}
```

Применение пространств имен

- Для исключения пересечения имен у различных разработчиков (в явном виде практически не используется)

- Для реализации понятия «пакет» - совокупности классов и др. сущностей, выполняющих общую задачу

Преобразование типов

Стандартное (C-style) преобразование типов позволяет:

- Преобразовывать стандартные типы
- Преобразовывать вверх по иерархии классов
- Преобразовывать неконсту к константе

не позволяет:

- Преобразовывать вниз по иерархии классов
- Преобразовывать константу к неконстанте
- Преобразовывать произвольные типы друг к другу

Преобразование топов: синтаксис

```
type static_cast <type> (expr)
//обычное приведение типов

type const_cast <type> (expr)
//приведение константы к неконстанте

type dynamic_cast <type> (expr)
//приведение вниз по иерархии. Если expr – указатель
//то в случае ошибки возвращется 0, если ссылка –
//выбрасывается исключение

type reinterpret_cast <type> (expr)
//произвольное преобразование типов (работает только
//с указателями)
```

Механизм RTTI (Run-Time Type Information)

Позволяет на стадии исполнения работать с типом объекта, в частности определять его и преобразовывать по иерархии наследования
В него входят:

dynamic_cast <type> (expr)

#include <typeinfo>
type_info typeid (type)
type_info typeid (expr)

RTTI работает на основе таблицы виртуальных функций

Обработка ошибок

Способы сообщения об ошибке из метода/функции:

- Если функция сама по себе не возвращает значения, то можно возвращать код ошибки
- Если функция возвращает значение, то об ошибке можно сообщать, возвращая невалидное значение
- Если такого значения нет, можно код ошибки возвращать через параметр

- Можно использовать механизм исключений

Исключения

```
class NullDivException {/*...*/};
double divide (double a, double b) throw NullDivException{
        if (b == 0) throw NullDivException ();
        return a/b;
}
int main (){
        try {
                cout << “a/b” = divide (a, b);
        }
        catch (NullDivException&){
                cerr << “Division by zero!”;
        }
        return 0;
}
```