Стурктуры

Строки

`Zero-base` строки или Си строки.

Другое представление строк.

```
struct String {
  char *data;
  int   len;
};
```

Полюсы, минусы.

Массивы

Списки

Односвязный, двусвязный.

Связный список - базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки (“связки”) на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком

расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Деревья

Дерево - одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы. Большинство источников также добавляют условие на то, что рёбра графа не должны быть ориентированными. В дополнение к этим трём ограничениям, в некоторых источниках указывается, что рёбра графа не должны быть взвешенными.

Поддерево - часть древообразной структуры данных, которая может быть представлена в виде отдельного дерева. Любой узел дерева `T` вместе со всеми его узлами-потомками является поддеревом дерева `T`. Для любого узла поддерева либо должен быть путь в корневой узел этого поддерева, либо сам узел должен являться корневым. То есть поддерево связано с корневым узлом целым деревом, а отношения поддерева со всеми прочими узлами определяются через понятие соответствующее поддерево (по аналогии с термином “соответствующее подмножество”).

Пошаговый перебор элементов дерева по связям между узлами-предками и узлами-потомками называется обходом дерева. Зачастую операция может быть выполнена переходом указателя по отдельным узлам. Обход, при котором каждый узел-предок просматривается прежде его потомков, называется предупорядоченным обходом или

обходом в прямом порядке (`pre-order walk`), а когда просматриваются сначала потомки, а потом предки, то обход называется поступорядоченным обходом или обходом в обратном порядке (`post-order walk`). Существует также симметричный обход, при котором посещается сначала левое поддерево, затем узел, затем - правое поддерево, и обход в ширину, при котором узлы посещаются уровень за уровнем (`N`-й уровень дерева — множество узлов с высотой `N`). Каждый уровень обходится слева направо.

Ассоциативные массивы

Ассоциативный массив - абстрактный тип данных (интерфейс к хранилищу данных), позволяющий хранить пары вида “(ключ, значение)” и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу:

- `INSERT`(ключ, значение)
- `FIND`(ключ)
- `REMOVE`(ключ)

Предполагается, что ассоциативный массив не может хранить две пары с одинаковыми ключами.

В паре `(k, v)` значение `v` называется значением, ассоциированным с ключом `k`. Семантика и названия вышеупомянутых операций в разных реализациях ассоциативного массива могут отличаться.

Операция `FIND`(ключ) возвращает значение, ассоциированное с

заданным ключом, или некоторый специальный объект `UNDEF`, означающий, что значения, ассоциированного с заданным ключом, нет. Две другие операции ничего не возвращают (за исключением, возможно, информации о том, успешно ли была выполнена данная операция).

Ассоциативный массив с точки зрения интерфейса удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов - например, строки.

Указанные три операции часто дополняются другими. Наиболее популярные расширения включают следующие операции:

- `CLEAR` - удалить все записи
- `EACH` - “пробежаться” по всем хранимым парам
- `MIN` - найти пару с минимальным значением ключа
- `MAX` - найти пару с максимальным значением ключа

В последних двух случаях необходимо, чтобы на ключах была определена операция сравнения.

В реализациях, основанных на хэш-таблицах, среднее время оценивается как `O(1)`, что лучше, чем в реализациях, основанных на деревьях поиска. Но при этом не гарантируется высокая скорость выполнения отдельной операции: время операции `INSERT` в худшем случае оценивается как `O(n)`. Операция `INSERT` выполняется долго, когда коэффициент заполнения становится высоким и необходимо

перестроить индекс хэш-таблицы.

Пример реализации `Hash` таблицы.

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>

struct Entry {
    char *key;
    char *value;
    struct Entry *next;
};
typedef struct Entry Entry;

struct HashTable {
    int size;
    struct Entry **table;
};
typedef struct HashTable HashTable;
```

Определение структур хранилища ключа и значений, хеш-таблицы.

```
HashTable *ht_create( int size ) {
    HashTable *hashtable = 0;
    int i;
```

```
    if( size < 1 ) return 0;

    if( ( hashtable = malloc( sizeof( HashTable ) ) ) == 0 )
{
        return 0;
    }

    if( ( hashtable->table = malloc( sizeof( Entry * ) * size
 ) ) == 0 ) {
        return 0;
    }
    for( i = 0; i < size; i++ ) {
        hashtable->table[i] = 0;
    }

    hashtable->size = size;
    return hashtable;
}
```

Функция создания и инициялизации хеш-таблицы

```
int ht_hash( HashTable *hashtable, char *key ) {
    unsigned long int hashval;
    int i = 0;

    while( hashval < ULONG_MAX && i < strlen( key ) ) {
```

```
        hashval = hashval << 8;
        hashval += key[ i ];
        i++;
    }

    return hashval % hashtable->size;
}
```

Функция расчета хеша по ключу

```
Entry *ht_newpair( char *key, char *value ) {
    Entry *newpair;

    if( ( newpair = malloc( sizeof( Entry ) ) ) == 0 ) {
        return 0;
    }

    if( ( newpair->key = strdup( key ) ) == 0 ) {
        return 0;
    }

    if( ( newpair->value = strdup( value ) ) == 0 ) {
        return 0;
    }

    newpair->next = 0;
    return newpair;
```

```
}
```

Функция создания хранилища ключа

```
void ht_set( HashTable *hashtable, char *key, char *value ) {
    int bin = 0;
    Entry *newpair = 0;
    Entry *next = 0;
    Entry *last = 0;

    bin = ht_hash( hashtable, key );
    next = hashtable->table[ bin ];
    while( next != NULL && next->key != 0 && strcmp( key, nex
t->key ) > 0 ) {
        last = next;
        next = next->next;
    }

    if( next != 0 && next->key != 0 && strcmp( key, next->key
 ) == 0 ) {
        free( next->value );
        next->value = strdup( value );

    } else {
        newpair = ht_newpair( key, value );

        if( next == hashtable->table[ bin ] ) {
```

```
            newpair->next = next;
            hashtable->table[ bin ] = newpair;
    } else if ( next == 0 ) {
            last->next = newpair;
        } else  {
            newpair->next = next;
            last->next = newpair;
        }
    }
}
```

Функция добавления элемента.

```
char *ht_get( HashTable *hashtable, char *key ) {
    int bin = 0;
    Entry *pair;

    bin = ht_hash( hashtable, key );
    pair = hashtable->table[ bin ];
    while ( pair != 0 && pair->key != 0 && strcmp( key, pair-
>key ) > 0 ) {
        pair = pair->next;
    }

    if ( pair == 0 || pair->key == 0 || strcmp( key, pair->ke
y ) != 0 ) {
        return 0;
```

```
    } else {
        return pair->value;
    }
}
```

Функция получения элемента.

Итераторы

Продолжения