Списки

Массив

Массив – последовательность элементов одного типа, расположенных в памяти друг за другом.

Преимущества и недостатки массива объясняются стратегией выделения памяти: память под все элементы выделяется в одном блоке.

“+” Минимальные накладные расходы.

“+” Константное время доступа к элементу.

“–” Хранение меняющегося набора значений.

Связный список

Связный список, как и массив, хранит набор элементов одного типа, но используется абсолютно другую стратегию выделения памяти: память под каждый элемент выделяется отдельно и лишь тогда, когда это нужно.

Связный список – это набор элементов, причем каждый из них является частью узла, который также содержит ссылку на [узел. Седжвик (с)] следующий и/или предыдущий узел списка.

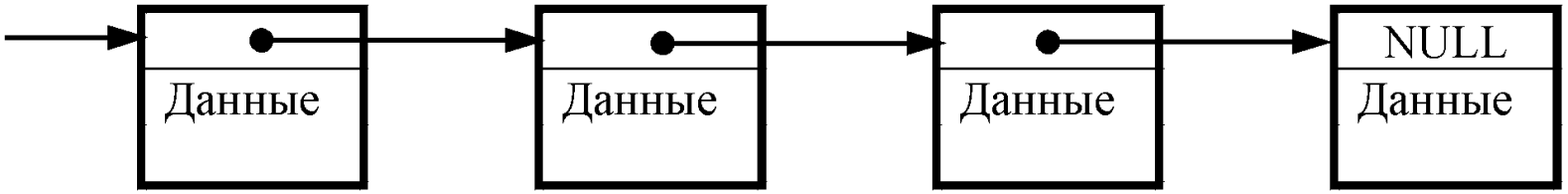
Связный список

Узел (*элемент списка*) – единица хранения данных, несущая в себе ссылки на связанные с ней узлы.

Узел обычно состоит из двух частей

– информационная часть (данные);

– ссылочная часть (связь с другими узлами).



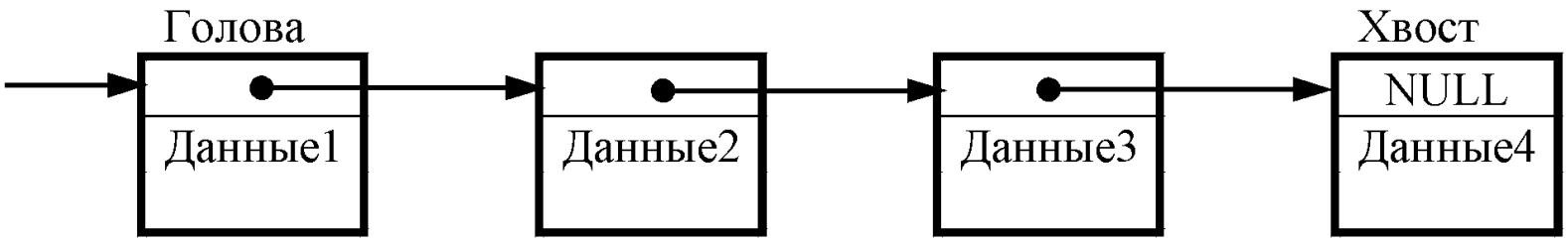
Связный список

Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов.

За эту гибкость приходиться жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в от слеживании связей от начала списка.

Линейный односвязный список

Линейный односвязный список – структура данных, состоящая из узлов, каждый из которых ссылается на следующий узел списка.



Линейный односвязный список

Узел, на который нет указателя, является первым элементом списка. Обычно этот узел называется *головой списка*.

Последний элемент списка никуда не ссылается (ссылается на NULL). Обычно этот узел называется *хвостом списка*.

Линейный односвязный список

Свойства односвязного списка

- Передвигаться можно только в сторону конца списка.
- Узнать адрес предыдущего элемента, опираясь только на содержимое текущего узла, нельзя.

Линейный односвязный список

Базовые операции

- Добавить элемент в начало или конец списка.
- Найти указанный элемент.
- Добавить новый элемент до или после указанного.
- Удалить элемент.

Элемент списка

```
struct person_t
{
    const char *name;
    int born_year;

    struct person_t *next;
};
```

```
typedef struct person_t person_t;

struct person_t
{
    const char *name;
    int born_year;

    person_t *next;
};
```

Создание/удаление узла списка

```
struct person_t* person_create(const char *name, int born_year)
{
    struct person_t *pers = malloc(sizeof(struct person_t));

    if (pers)
    {
        pers->name = name;
        pers->born_year = born_year;
        pers->next = NULL;
    }

    return pers;
}

void person_free(struct person_t *pers)
{
    free(pers);
}
```

Добавление элемента в список

```
void list_add_front_usual(struct person_t **head,
                                        struct person_t *pers)
{
    pers->next = *head;
    *head = pers;
}
```

NB: функции, изменяющие список, должны возвращать указатель на новый первый элемент.

Добавление элемента в список

```
struct person_t* list_add_front(struct person_t *head,
                                struct person_t *pers)
{
    pers->next = head;
    return pers;
}
```

Использование

```
head = add_front(head, pers);
```

Добавление элемента в список

```
struct person_t* list_add_end(struct person_t *head,
                                        struct person_t *pers)
{
    struct person_t *cur = head;

    if (!head)
        return pers;

    for ( ; cur->next; cur = cur->next)
        ;

    cur->next = pers;

    return head;
}
```

Добавление элемента в список

Добавление элемента в конец нашего простого списка – операция порядка O(N). Чтобы добиться времени O(1), можно завести отдельный указатель на конец списка.

```
struct list_t
{
    struct persont_t *head;
    struct persont_t *tail;
};
```

Поиск элемента в списке

```
struct person_t* list_lookup(struct person_t *head,
                                              const char *name)
{
    for ( ; head; head = head->next)
        if (strcmp(head->name, name) == 0)
            return head;

    return NULL;
}
```

Поиск занимает время порядка O(N) и эту оценку не улучшить.

Обработка всех элементов списка

```
void list_apply(struct person_t *head,
                void (*f)(struct person*, void*),
                                          void *arg)
{
    for ( ; head; head = head->next)
        f(head, arg);
}
```

- **head**: список
- **f**: указатель на функцию, которая применяется к каждому элементу списка
- **arg**: аргумент функции f

Обработка всех элементов списка

```
// печать информации из элемента списка
void person_print(struct person *pers, void *arg)
{
    char *fmt = arg;
    printf(fmt, pers->name, pers->born_year);
}

// list_apply(l1, person_print, "l1: %s %d\n");

// подсчет количества элементов списка
void person_count(struct person *pers, void *arg)
{
    int *counter = arg;
    (*counter)++;
}

// list_apply(l2, person_count, &n); // где int n = 0;
```

Освобождение списка

Так делать НЕЛЬЗЯ! Почему?

```
void list_free_all(struct person_t *head)
{
    for ( ; head; head = head->next)
        person_free(head);
}
```

Освобождение списка

```
void list_free_all(struct person *head)
{
    struct person *next;

    for ( ; head; head = next)
    {
        next = head->next;
        person_free(head);
    }
}
```

Наша функция free_all не освобождает память из поля name (см. person_create).

Удаление элемента по имени

```
struct person* del_by_name(struct person *head,
                                          const char *name)
{
    struct person *cur, *prev = NULL;

    for (cur = head; cur; cur = cur->next)
    {
        if (strcmp(cur->name, name) == 0)
        {
            if (prev)
                prev->next = cur->next;
            else
                head = cur->next;
            free(cur);
            return  head;
        }
        prev = cur;
    }

    return NULL;
}
```

У этой реализации есть недостаток, который не замечали лет пять. Какой?

Списки: дальнейшее развитие

- Представление элемента списка
  - Универсальный элемент (void*).
- Двусвязные списки
  - Требует больше ресурсов.
  - Поиск последнего и удаление текущего – операции порядка O(1).