Лекция 6: Рекусивные структуры данных и динамическая память в языке Си

Д. А. Караваев

Санкт-Петербургский государственный университет телекоммуникаций им. проф. М. А. Бонч-Бруевича

Факультет РТС, Кафедра РОС

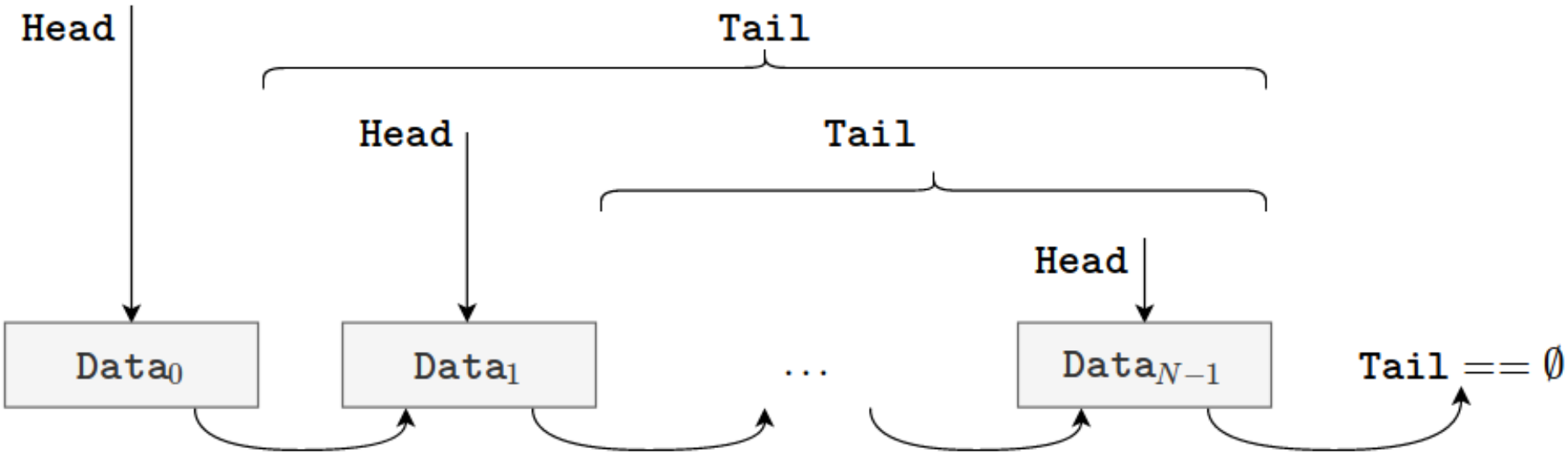
Факультатив «Программирование в ЦОС»

Осень 2019

25.11.2019 Санкт-Петербург

Связные списки

Простейшим примером *рекурсивной* структурой данных является **однонаправленный связаный список**, который состоит из *головы* и *хвоста*, который также является связным списком, либо отсутствует.



Реализация в язык Си

Односвязный список можно реализовать через структуру, в которой содержится *указатель* на хвост.

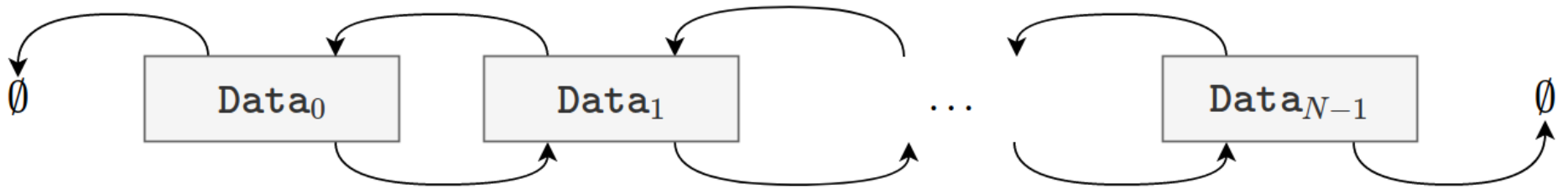
```c
/* Определение списка: */
typedef struct list_impl
{
    int data; /* Содержимое головы. */
    struct list_impl* tail; /* Хвост. */
} list_impl_t;
/* Создание головы: */
list_impl_t head = {.data = 100, .tail = NULL}; /* Нулевой указатель! */
/* Создание хвоста: */
list_impl_t tail = {.data = 200, .tail = NULL};
/* Добавление хвоста: */
head.tail = &tail;
```

Инкапсуляция работы со списком

```
/* Управляющая структура с головой списка (Данные): */
typedef struct
{
    list_impl_t* head; /* Хвост. */
    size_t       size;  /* Число элементов в списке. */
} list_t;
/* Методы: */
/* Добавить элемент в конец списка: */
void list_emplace_back(list_t* list, int data);
/* Подсчитать число элементов в списке с данным значением: */
size_t list_count(const list_t* list, int data);
/* Узнать значения по индексу: */
int list_at(const list_t* list, size_t index);
/* Удалить элемент из списка по индексу: */
void list_delete(list_t* list, size_t index);
```

Двунаправленный связный список

```
typedef struct list_impl
{
    int data; /* Содержимое головы. */
    struct list_impl* succ; /* Последующий элемент. */
    struct list_impl* pred; /* Предыдущий элемент. */
} list_impl_t;
/* Создание списка: */
list_impl_t first = {.data = 100, .succ = NULL, .pred = NULL};
list_impl_t second = {.data = 200, .succ = NULL, .pred = &first};
first.succ = &second;
```



Замечания по поводу работы со списками

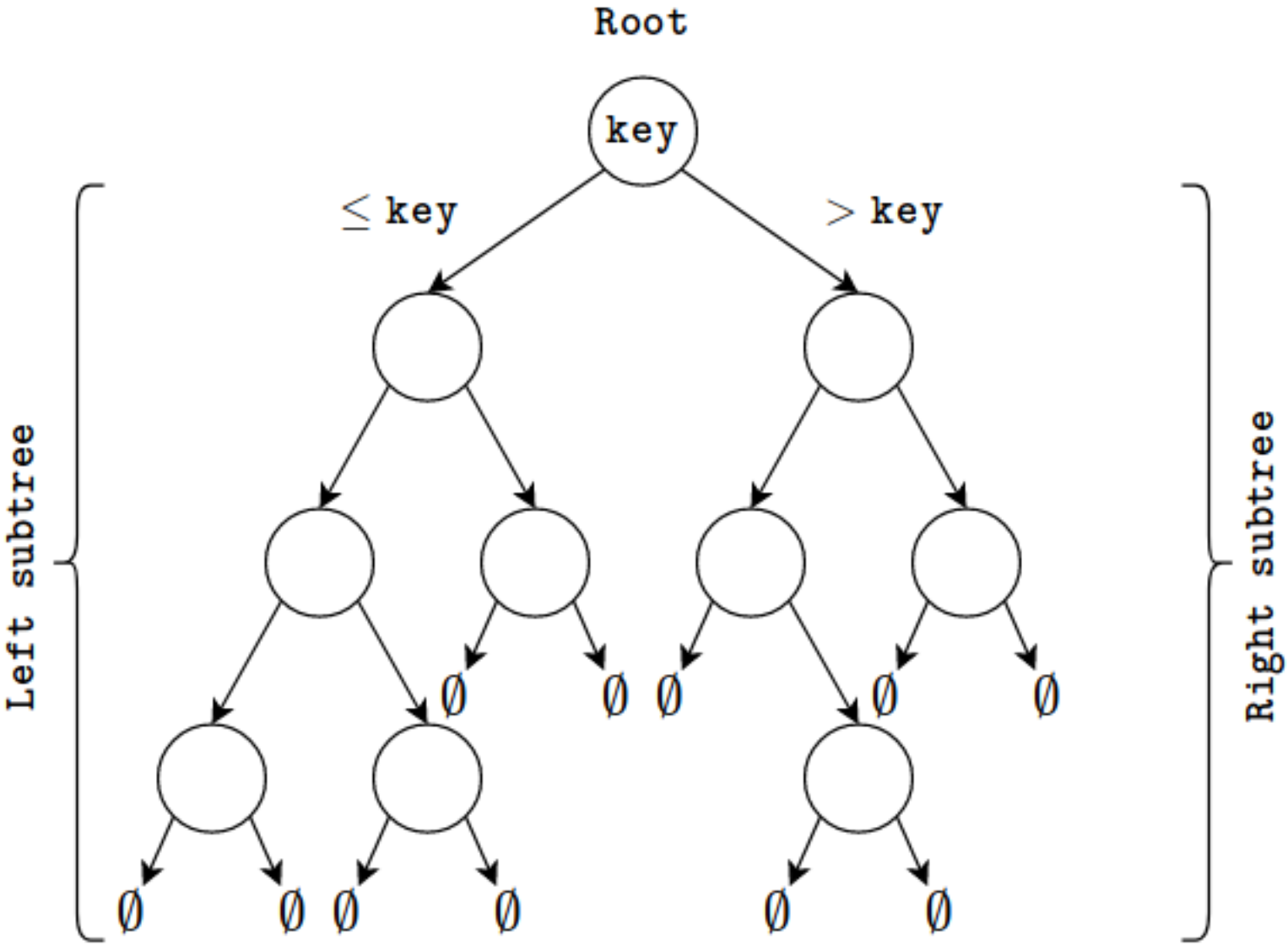
1. **Алгоритмическая сложность**: Большинство операций в списках имеют $O(N)$ или $\Theta(N)$ временную сложность;
2. **Реализация стека и очереди**: Списки удобно использовать для реализации стека (или очереди), так как число элементов не ограничено;
3. **Операции над двунаправленном списком**: Можно опредлить большое множество полезных операций над двунаправленном списком.
4. **Хранения "тяжелых" структур**: Списки удобны для хранения структур (пользовательских типов) с большим количеством полей, к которым нет необходимости применять индексацию.

Дерево бинарного поиска

Другим примером рекурсивной структуры данных служит **дерево**, которое состоит из **корня** (родителя), в котором хранится значение (*ключ*) и у которого может быть несколько **поддеревьев** (потомков). Деревья, у которых нет поддеревьев, называются **листьям**.

Если у дерева может быть не более двух поддеревьев, и *для каждого родительского ключа ключи большее него храняться в правом поддереве, а меньше в левом*, то таком дерево называется **дерево бинарного поиска**. В нашем случае, все значения в дереве *уникальны*.

Иллюстрация дерева бинарного поиска



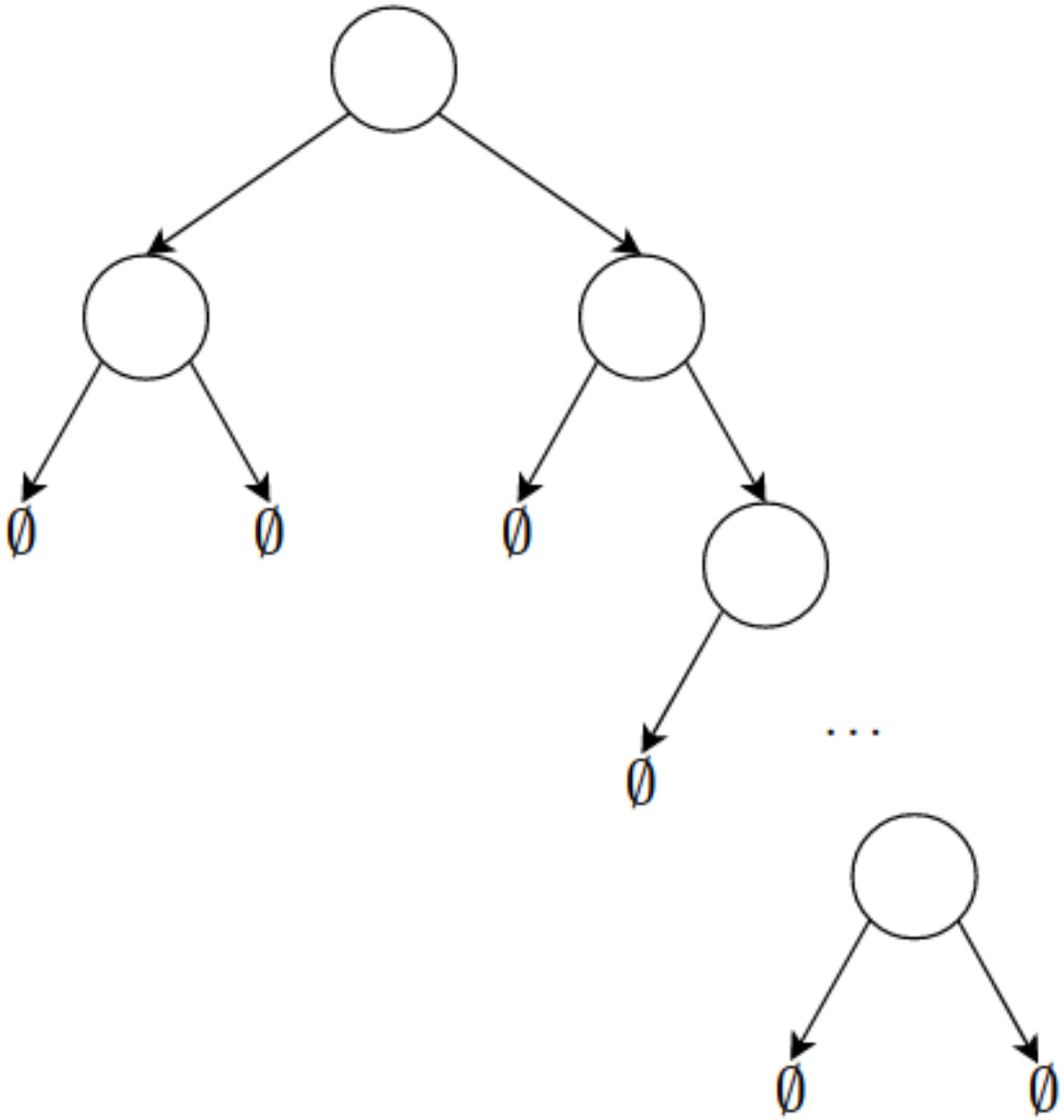
Представление дерева бинарного поиска в языке Си

```c
typedef struct bstree_node
{
    int key; /* Ключ. */
    struct bstree_node* left; /* Левое поддерево. */
    struct bstree_node* rigth; /* Правое поддерево. */
} bstree_node_t;
/* Управляющая структура с корнем дерева: */
typedef
{
    bstree_node_t* root; /* Корень дерева. */
    size_t size; /* Размер дерева. */
} bstree_t;
void bstree_emplace(bstree_t* tree, int key); /* Добавить элемент. */
int bstree_find(const bstree_t* tree, int key); /* Найти элемент. */
void bstree_delete(bstree_t* tree, int key); /* Удалить элемент. */
```

Замечания к дереву бинарного поиска

1. **Алгоритмическая сложность**: Когда, дерево заполнено равномерно алгоритмеческая сложность всех операций пропорциональна его высоте $O(h) = O(\log N)$, где N - число элементов в дереве.
2. **Расбаланисровка**: Яление, при котором основная "масса" элементов дерева сосредоточена вдоль одного края. Тогда дерево *вырождается* в однонаправленный список и преимущество в скорости поиска элементов теряется. Для борьбы с этим явлением используют более сложные реализации бинарных деревьев (красно-черные и AVL деревья).
3. **Ассоциативный контейнер**: Деревья часто используются для реализации *ассоциативного контейнера (словарь)*, в котором тип индекса (*ключа*) может быть любым упорядоченным типом.

Иллюстрация расбалансировки



Реализации вставки в односвязный список

```
void list_emplace_back(list_t* list, int data)
{
    return list_emplace_back_impl(list->head, data);
}
void list_emplace_back_impl(list_impl_t* head, int data)
{
    if (head->tail == NULL)
    {
        list_impl_t tail = {.data = data, .tail = NULL};
        head->tail = &tail; /* Ошибка. tail - локальная переменная! */
        return;
    }

    return list_emplace_back_impl(head->tail, data);
}
```

Спасибо за внимание!