

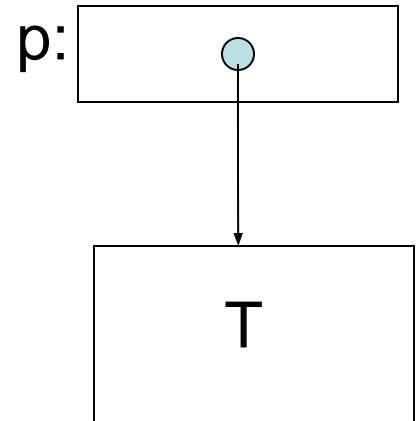
Распределение памяти

Время существования объекта:

- Глобальные объекты – существующие всё время исполнения программы
- Локальные объекты процедур – существующие во время исполнения функции
- Динамические объекты – появляются по специальному запросу в «куче»
 - исчезают по специальному запросу
 - исчезают автоматически после того, как становятся недоступными (автоматическая сборка мусора)

Создание и удаление нового объекта (Pascal)

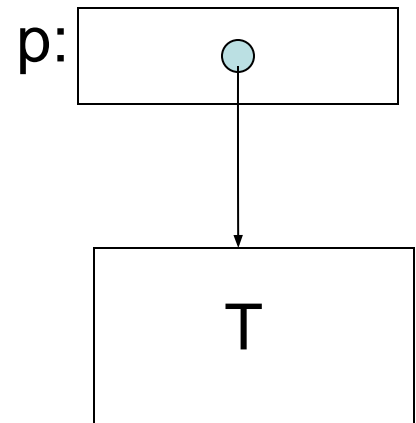
```
var p : ^T;  
...  
new(p);  
...  
dispose(p)
```



- `new(p)`
 - размещается память для хранения объекта типа T
 - указателю p присваивается ссылка на новый анонимный объект
- `dispose(p)`
 - память освобождается для дальнейшего переиспользования

Создание и удаление нового объекта (C)

```
// Полезный макрос  
#define new(x) x = malloc(sizeof(*x))  
T * p;  
...  
new(p);  
...  
free(p);
```

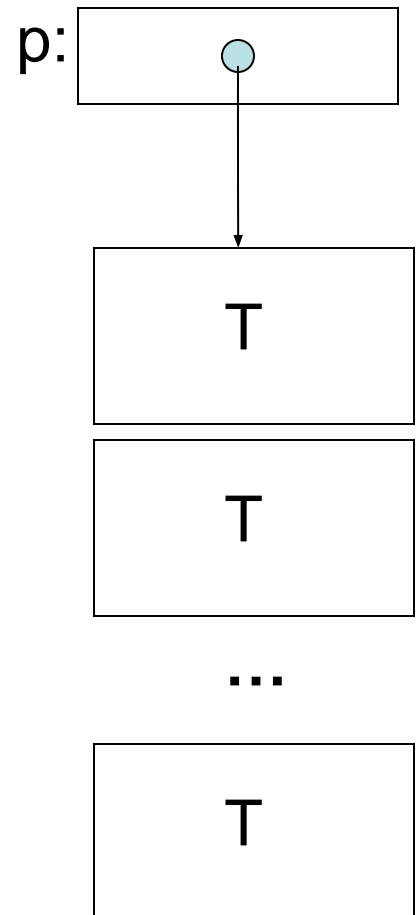


- `new(p)`
 - размещается память для хранения объекта типа T
 - указателю p присваивается ссылка на новый анонимный объект
- `free(p)`
 - память освобождается для дальнейшего переиспользования

Создание и удаление массива (C)

```
// Полезный макрос
#define newarray(a,n) a = calloc(sizeof(*p),n)
T * p;
...
newarray(p, N);
...
free(p);
```

- `new(p)`
 - размещается память для хранения **N** объектов типа **T**
 - указателю `p` присваивается ссылка на **первый** новый анонимный объект
- `free(p)`
 - память освобождается для дальнейшего переиспользования



Накладные расходы

- Дополнительная память для организации кучи
- Сам указатель занимает место (возможно большее, чем размещаемый объект)

```
char * p;  
new(p);
```
- Управление размещением памятью – сложные операции
- Фрагментация – свободная память есть, но слишком мелкими кусками

Типичные ошибки

- `new(p);`
`q = p;`
`free(p);`
`free(q);`
- `new(p);`
`new(p);`
- `p = NULL;`
`free(p);`
- `p = &x;`
`free(p);`
- `newarray(p, 10);`
`p++;`
`free(p);`
- `new(p);`
`p->a = 5;`
`free(p);`
`if (p->a == 5) ...`

Автоматическая сборка мусора

- Можно считать, что динамически размещённый объект существует до конца исполнения программы
- Позволяет избежать большинство самых «трудных» ошибок
 - проявляются не всегда
 - не воспроизводятся
 - проявляются далеко от места ошибки и не связаны явно с распределением памяти
- Доступна в Lisp, Oberon, Visual Basic, C#, и эффективна

Автоматическая сборка мусора

- Всё-таки весьма сложна
- Случается в непредсказуемые моменты времени

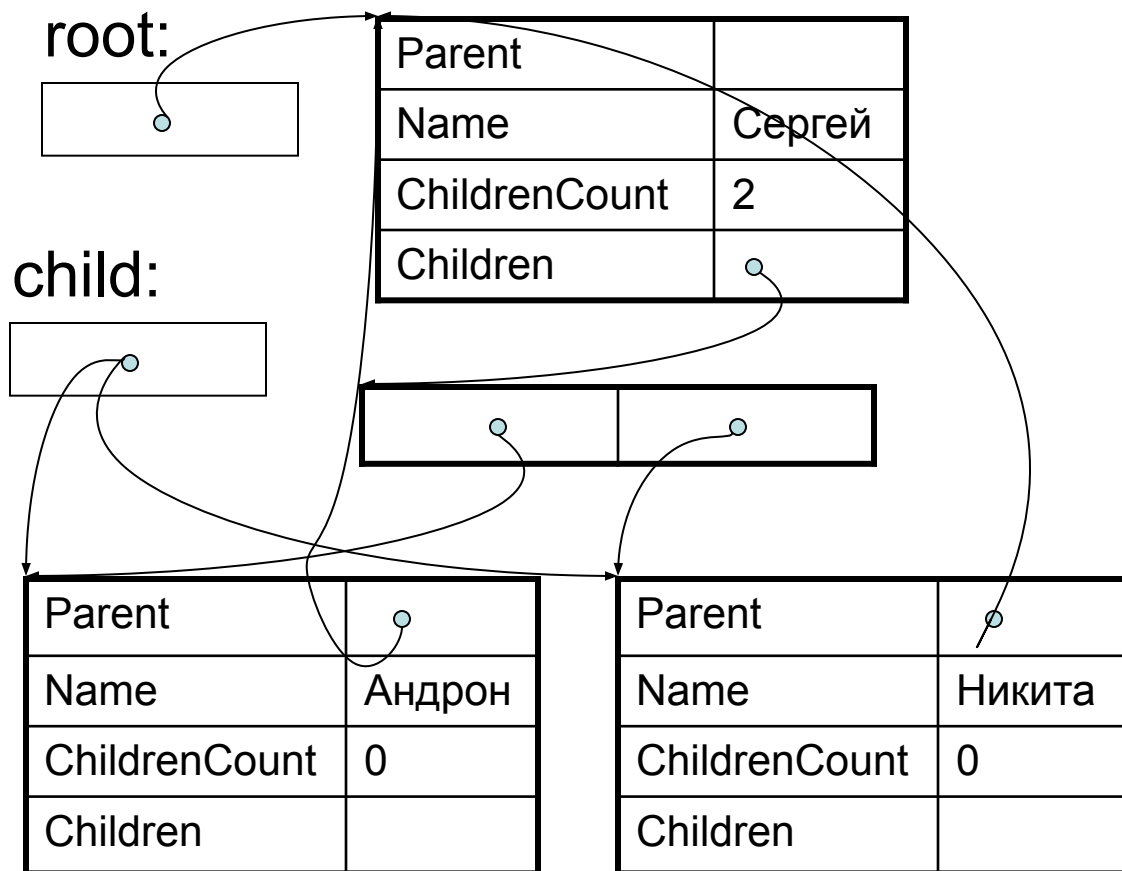
Пример - дерево

- В корне дерева хранится информация об имени
- Для каждого узла дерева – ссылка на родителя
- Для каждого узла дерева – ссылки на произвольное количество детей

Пример

```
struct Person
{
    struct Person * Parent;
    char Name[32];
    unsigned int ChildrenCount;
    struct Person * Children;
} * root, * child;

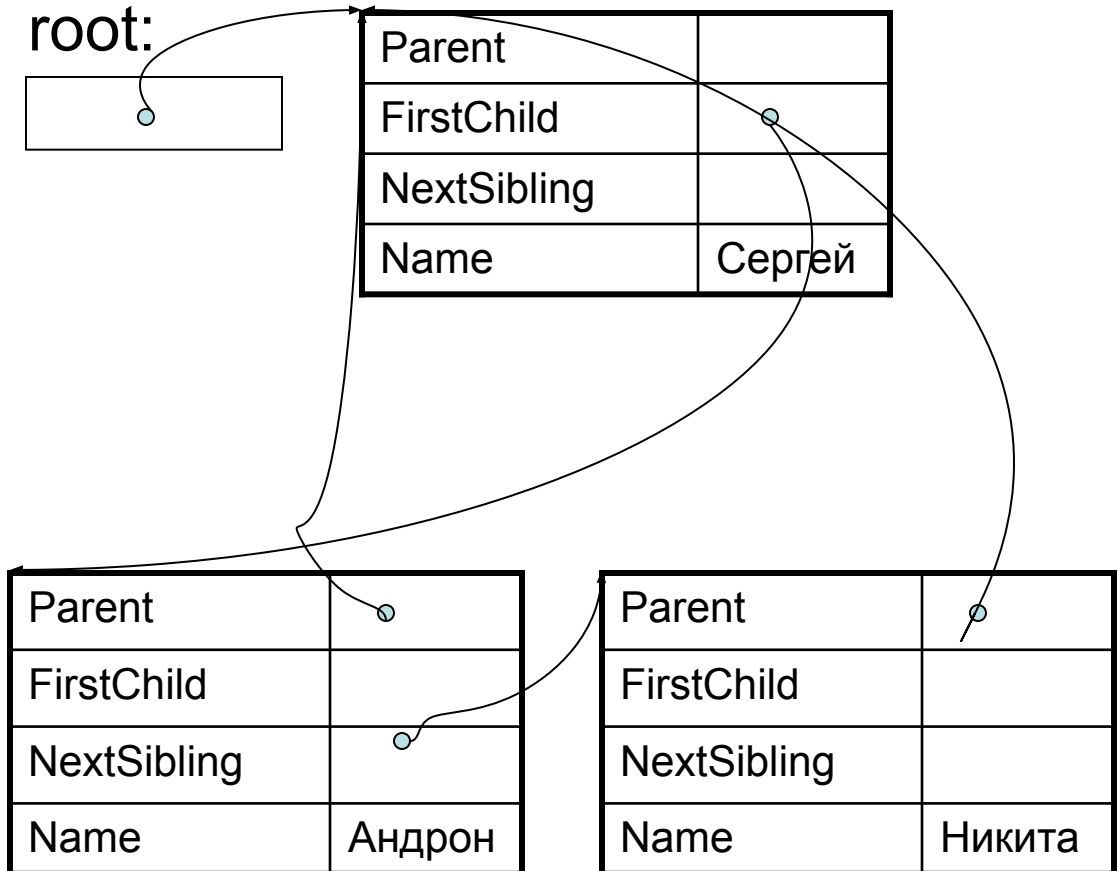
new(root);
root->Parent = NULL;
strcpy(root->Name, "Сергей");
root->ChildrenCount = 2;
newarray(root->Children, 2);
child = new(root->Children[0]);
strcpy(child->Name, "Андрон");
child->ChildrenCount = 0;
child->Children = NULL;
child->Parent = root;
child = new(root->Children[1]);
strcpy(child->Name, "Никита");
child->ChildrenCount = 0;
child->Children = NULL;
child->Parent = root;
```



Цикл: `root->Children[1].Parent == root`

Пример – кодирование двоичным деревом

```
struct Person
{
    struct Person * Parent,
        * FirstChild,
        * NextSibling;
    char Name[32];
} * root;
```



Сравнение

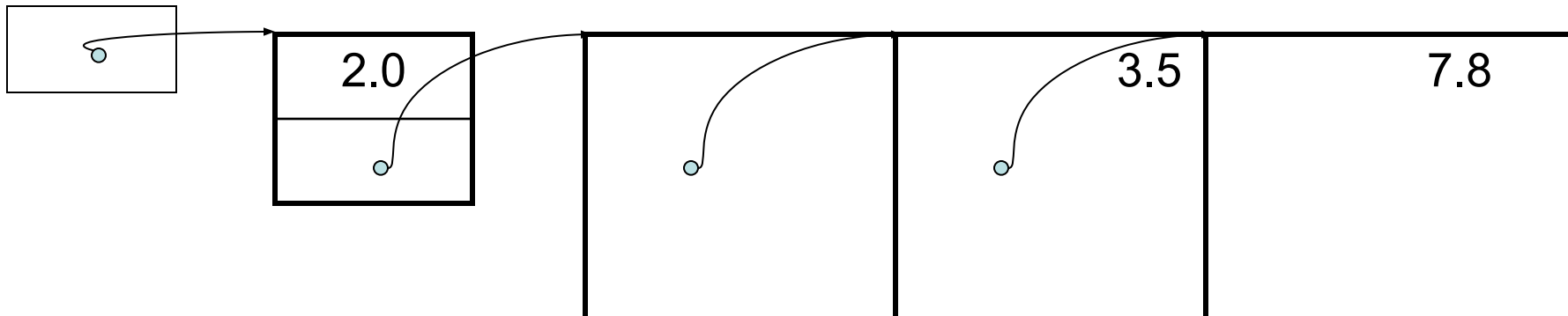
- ChildrenCount,
Children
 - Накладные расходы:
 $14N - 4$
 - Быстрый способ
узнать количество
детей
 - Быстрый доступ к
любому поддереву
- FirstSibling,
NextSibling
 - Накладные расходы
 $12N$
 - Простота вставки
поддерева

Односвязные списки

```
typedef double T;  
  
typedef struct ListElem{  
    T info;  
    struct ListElem * next;  
} *List;  
  
List root;
```

- `info` – информация, содержащаяся в элементе списка
- `next` – ссылка на следующий элемент

root:



Односвязные списки - примеры

- Подсчёт количества элементов
- Сумма элементов списка

```
int cnt = 0;  
for (List p = root; p!=NULL; p=p->next)  
    ++ cnt;
```

```
T sum = 0;  
for (List p = root; p!=NULL; p=p->next)  
    sum += p->info;
```

Односвязные списки – стек (магазин, LIFO)

- Добавление элемента x в начало списка (push)
- Получение значения первого элемента (top)
- Удаление первого элемента (pop)
- Проверка пустоты стека (empty)

```
List tmp;  
new(tmp);  
tmp->info = x;  
tmp->next = root;  
root = tmp;
```

```
root->info
```

```
List tmp = root->next;  
free(root);  
root = tmp;
```

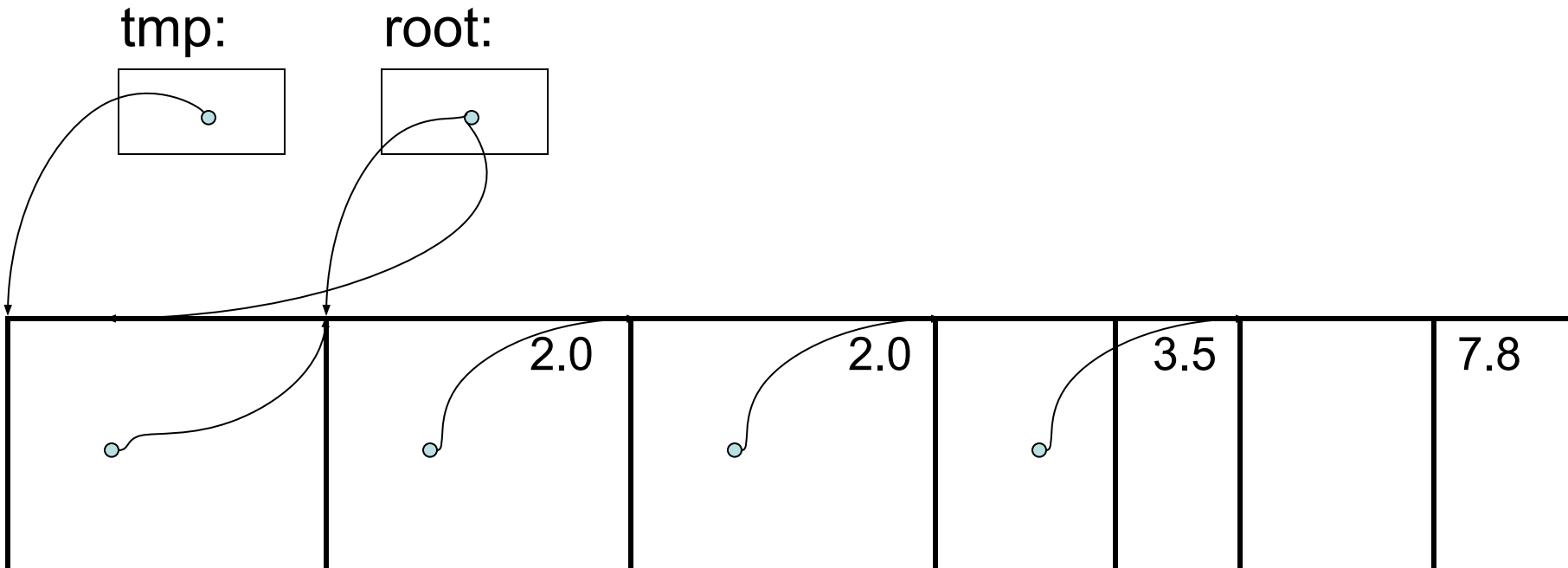
```
root == NULL
```

LIFO = Last In - First Out

Односвязные списки – стек (магазин, FIFO)

- Добавление элемента x в начало списка (push)

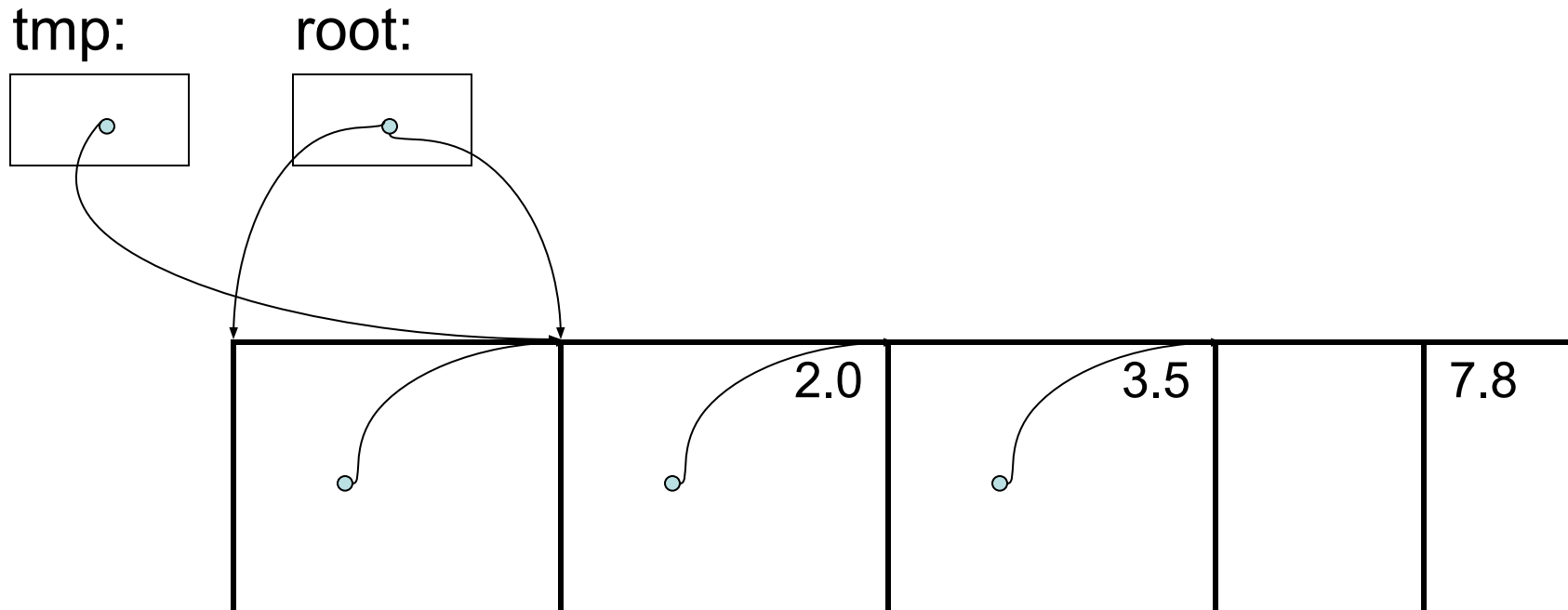
```
List tmp;  
new(tmp);  
tmp->info = x;  
tmp->next = root;  
root = tmp;
```



Односвязные списки – стек (магазин, FIFO)

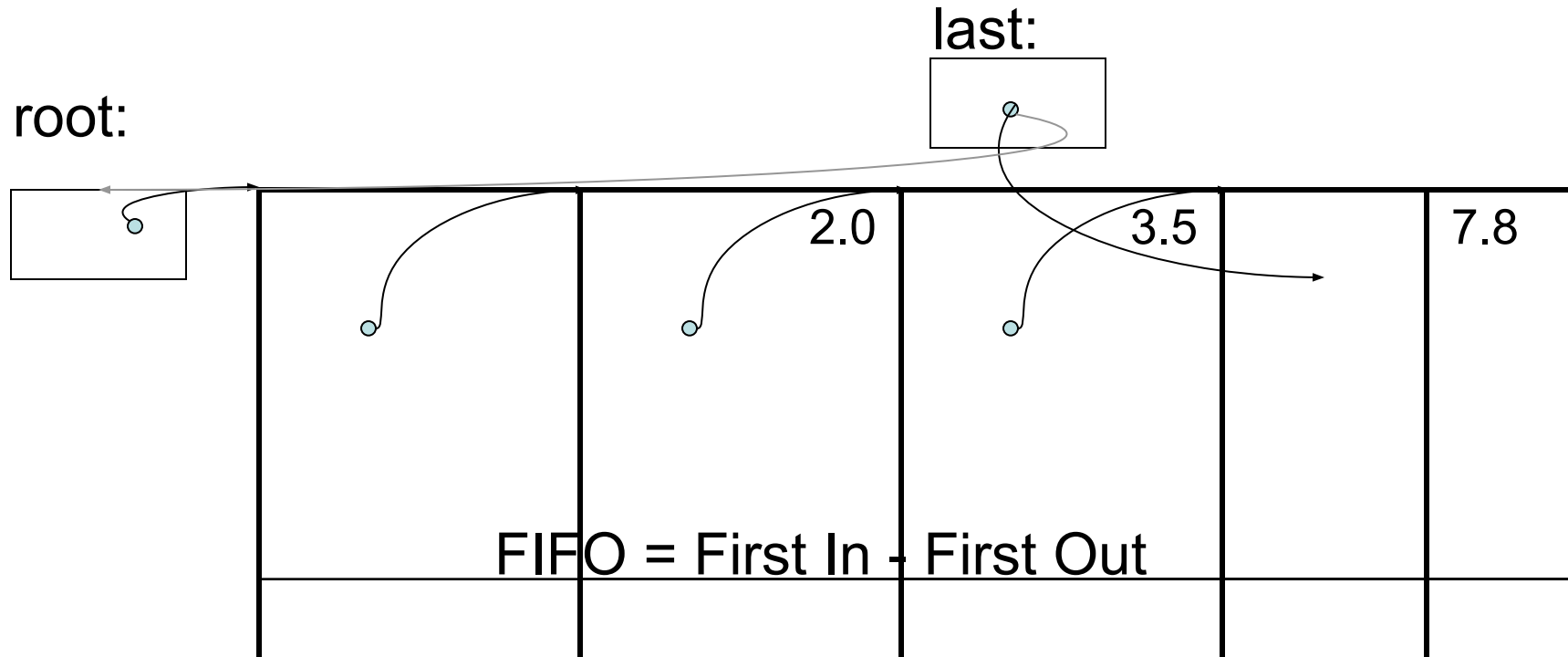
- Удаление первого элемента (pop)

```
List tmp = root->next;  
free(root);  
root = tmp;
```



Односвязные списки – очередь (FIFO)

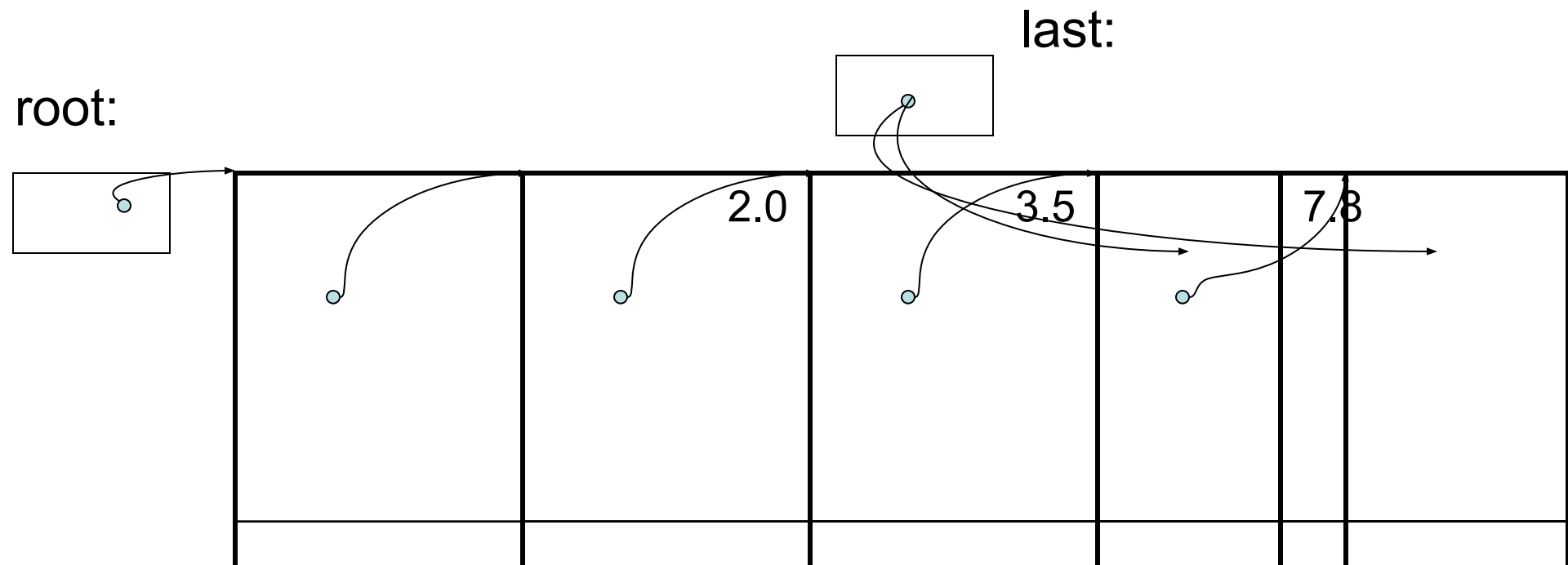
- Добавление элемента – в конец очереди
`List * last = &root;`
- «Обслуживание» - из начала очереди



Односвязные списки – очередь (LIFO)

- Добавление элемента x в конец списка

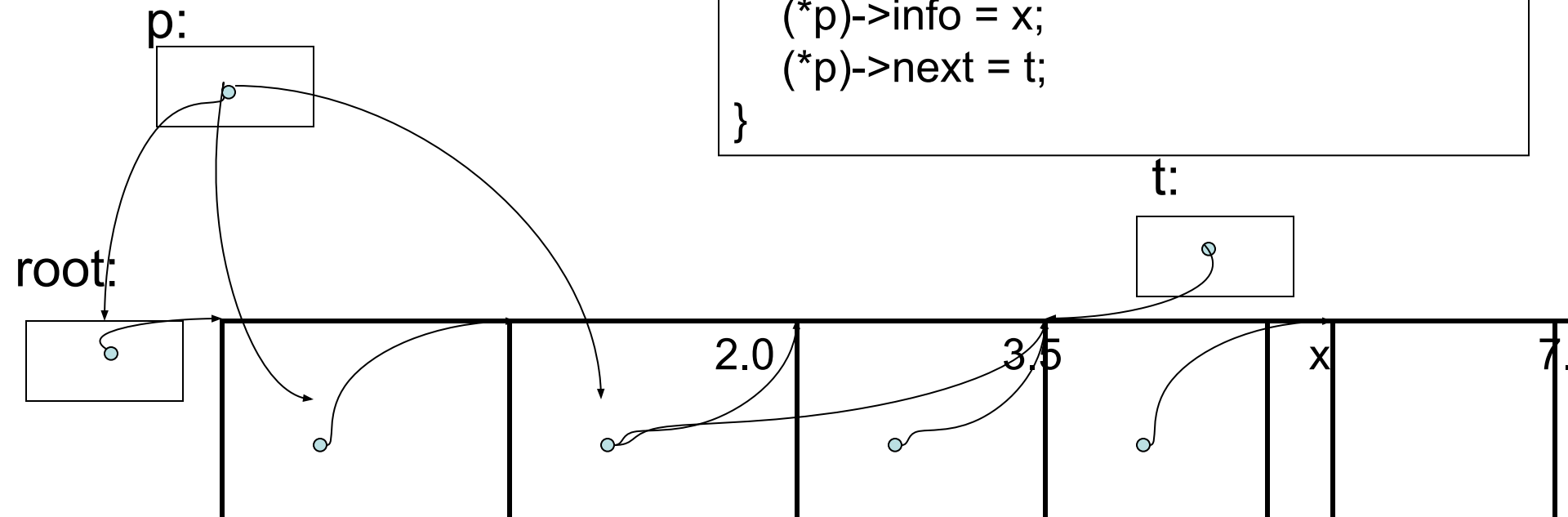
```
new(*last);  
(*last) -> info = x;  
(*last) -> next = NULL;  
last = &((*last)->next)
```



Упорядоченные односвязные СПИСКИ

- Добавление элемента x (равного 5.0)

```
List * p = & root;  
while (*p != NULL && (*p)->info < x)  
    p = &((*p)->next);  
if (*p == NULL || ((*p)->info != x))  
{  
    List t = *p;  
    new(*p);  
    (*p)->info = x;  
    (*p)->next = t;  
}
```

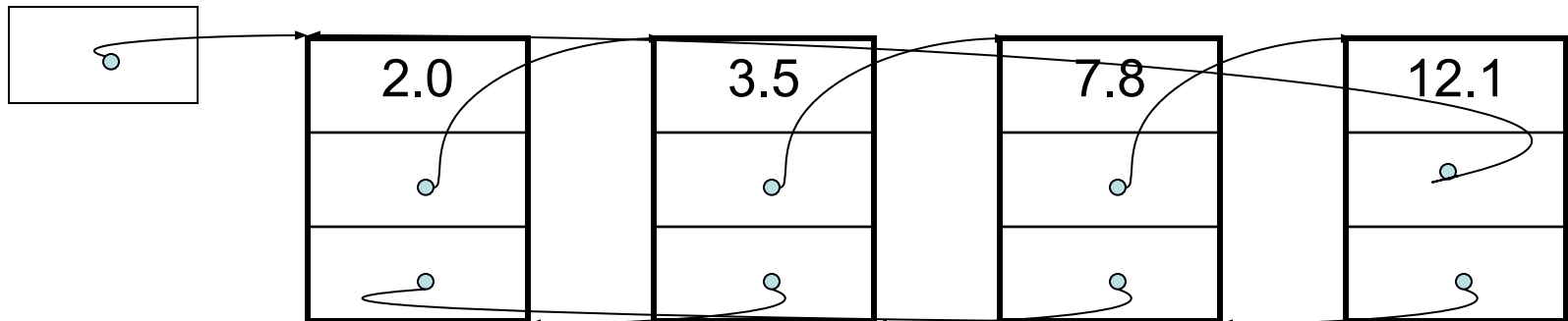


Двусвязные циклические списки

```
typedef double T;  
  
typedef struct ListElem{  
    T info;  
    struct ListElem * next, * prev;  
} *List;  
  
List root;
```

- Дополнительная ссылка на предыдущий элемент
- Следующий последнего – первый
- Предыдущий первого – последний

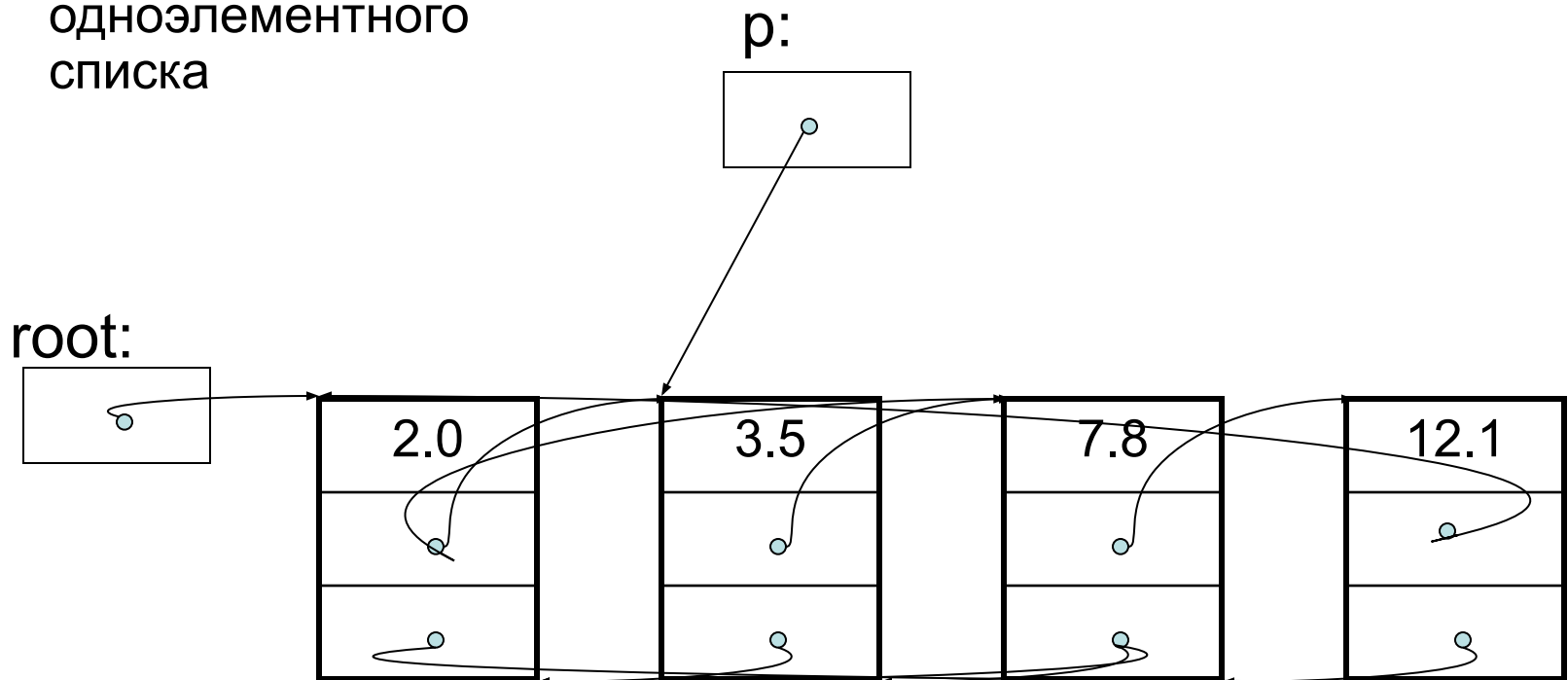
root:



Двусвязные циклические списки

- Удаление элемента, заданного ссылкой p
- Особо рассмотреть случай одноэлементного списка

```
p->prev->next = p->next;  
p->next->prev = p->prev;  
free(p);
```



Прочие структуры данных

- Двоичные деревья
 - ссылка на правое поддерево
 - ссылка на левое поддерево
- «Вагонная» память – оба конца списка равноправны:
 - добавлять
 - удалять
 - выбирать крайний