Абстрактный тип данных «список». Связный список. Односвязный список, двусвязный список. Основные операции и их вычислительная сложность.

- **Список** *(list)*– это абстрактный тип данных, представляющий собой набор однотипных элементов *(item, entry, node),* упорядоченных в соответствии с их позицией в списке– индексом *(index)*.
- Каждый элемент списка помимо позиции также хранит некоторое значение *(value)*.
- Нумерация элементов начинается с единицы или нуля.

Связный список (linked list) – это динамическая структура данных для хранения информации, в которой каждый элемент хранит указатели на один или несколько других элементов

Основные операции связного списка:

Операция	Описание	Вычислительная сложность	Сложность по памяти
`AddFront(L, x)`	Добавляет элемент *x* в начало списка L	O(1)	O(1)
`AddEnd(L, x)`	Добавляет элемент *x* в конец списка L	O(*n*)	O(1)
`Lookup(L, x)`	Отыскивает элемент *x* в списке L	O(*n*)	O(1)
`Size(L)`	Возвращает количество элементов в списке L	O(1) или O(*n*)	O(1)

Односвязный список (singly linked list)

- Размер списка заранее неизвестен – элементы добавляются во время работы программы (динамически)

- Память под элементы выделяется динамически (функции: malloc, calloc, realloc, free)*

```
struct listnode {
    char *key; // Ключ
    int value; // Значение
    struct listnode *next; // Указатель на следующий элемент (узел)
};

// Создание узла списка
struct listnode *list_createnode(char *key, int value) {
    struct listnode *p;
    p = malloc(sizeof(*p));
    if (p != NULL) {
        p->key = key;
        p->value = value;
        p->next = NULL;
    }
    return p;
}

// Добавление элемента в начало списка

    struct listnode *list_addfront(struct listnode *list, char *key, int
value)
{
    struct listnode *newnode;
    newnode = list_createnode(key, value);

    if (newnode != NULL) {
        newnode->next = list;
        return newnode;
    }
    return list;
}

// Поиск элемента в списке (lookup)

struct listnode *list_lookup(struct listnode *list, char *key)
{
    for ( ; list != NULL; list = list->next)
        if (strcmp(list->key, key) == 0)  return list;
    return NULL; // Не нашли
}
```

- Начиная с головы списка, поочередно просматриваем узлы и сравниваем ключи
- В худшем случае требуется просмотреть все узлы, это требует O(n) операций

```c
// Удаление элемента
struct listnode *list_delete(struct listnode *list, char *key)
{
    struct listnode *p, *prev = NULL;

    for (p = list; p != NULL; p = p->next) {
        if (strcmp(p->key, key) == 0) {
            if (prev == NULL)
                list = p->next;      // Удаляем голову
            else
              prev->next = p->next; // Есть элемент слева
            free(p);
            return list;
        }
        prev = p;
    }
    return NULL;
}
```

Вычислительная сложность операций в худшем случае:

Операция	Реализация списка	
	Массив	Односвязный список
$\textsc{Insert}(L, i, x)$	$\Theta(n)$	$\Theta(n)$
$\textsc{Lookup}(L, x)$	$\Theta(n)$	$\Theta(n)$
$\textsc{GetItem}(L, i)$	$O(1)$	$\Theta(n)$
$\textsc{Delete}(L, i)$	$\Theta(n)$	$\Theta(n)$
$\textsc{Next}(L, i)$	$O(1)$	$\Theta(n)$
$\textsc{Prev}(L, i)$	$O(1)$	$\Theta(n)$
$\textsc{Size}(L)$	$O(1)$	$\Theta(n)$

Двусвязные списки

- Каждый узел двусвязного списка имеет четыре поля: key, next,prev и value
- Поле key – некоторый ключ, ассоциированный с узлом
- В поле next хранится адрес узла, следующего за текущим, в поле prev – предшествующего

```
//Структура узла списка

struct dlistnode {

  char *key;           //Ключ

  struct dlistnode* next;  //Следующий  узел

  struct dlistnode* prev;  //Предыдущий узел

  int value; // Данные

};

// Создание нового узла

struct dlistnode *dlist_createnode(char *key, int value)
{
    struct dlistnode *p;
    p = malloc(sizeof(*p));
    if (p != NULL) {
        p->key = key;
        p->value = value;
        p->next = NULL;
        p->prev = NULL;
    }
    return p;
}

// Добавление узла в начало списка

struct dlistnode *dlist_addfront(struct dlistnode *list,
        char *key, int value)
{
    struct dlistnode *newnode;
    newnode = dlist_createnode(key, value);
    newnode->next = list;
    if (list != NULL) {
        list->prev = newnode;
        return newnode;
```

```
    }
    return list;
}
```

- Функция создает в памяти новый узел с заданным значением поля key
- В поле next нового узла заносится адрес головы списка
- Если список не пуст, необходимо записать в указатель prev первого узла адрес нового элемента

```
// Добавление узла в конец списка
struct dlistnode *dlist_addend(struct dlistnode *list,
            char *key, int value)
{
    struct dlistnode *newnode;
    newnode = dlist_createnode(key, value);

    if (list == NULL)
        return newnode;

    struct dlistnode *node = list;
    while (node->next != NULL)
        node = node->next;
    node->next = newnode;
    newnode->prev = node;
    return list;
}
```

- Функция создает в памяти новый узел с заданным значением поля key
- Выполняется проход по списку до последнего узла
- Указатель next последнего узла связывается с новым узлом
- Указатель prev нового узла связывается с последним узлом

```
// Удаление узла

struct dlistnode *dlist_delete(struct dlistnode *list, char *key)
{
    struct dlistnode *p;

    for (p = list; p != NULL; p = p->next) {
        if (strcmp(p->key, key) == 0) {
            if (p->prev == NULL)
                list = p->next;
            else
```

```
                p->prev->next = p->next;

            if (p->next != NULL)
                p->next->prev = p->prev;
            free(p);
            return list;
        }
    }
    return NULL;
}
```

- Идем по списку до тех пор, пока не найдем удаляемый узел
- Корректируем указатели prev и next следующего и предыдущего узла, если они существуют
- Освобождаем память

Вычислительная сложность операций:

- **Добавление в начало**: **Θ(1)**
- **Добавление в конец** : в худшем и среднем случаях **Θ(n)**,где **n**–это количество узлов в списке.
- **Поиск узла** : **Θ(n)**
- **Удаление узла**: в худшем случае **Θ(n).**
