Лекция 22
Тема: Структуры данных.
Списки.

Понятие «***структура данных***» является расширением понятия «***тип данных***».

Структуры данных можно разделить на ***простые*** и ***сложные***.

***Простые* структуры**

состоят из единственного элемента, несущего какую-то смысловую нагрузку. Этот элемент можно представить как совокупность байтов или бит, но каждый отдельно взятый байт или бит нельзя рассматривать в отрыве от других. В языке С++ простым структурам соответствует скалярный тип (целые, вещественные, символьные, булевский).

Сложные структуры

– это набор некоторым образом сгруппированных данных. В С++ сложным структурам соответствует понятие структурированного типа (массив, структура).

Типы структур

- *Однородные,* состоящие из элементов одного типа (массивы);
- *Неоднородные* – элементы которых могут иметь различный тип, например, структуры;
- *Последовательные* или *несвязанные* структуры данных. В таких структурах все элементы, входящие в ее состав, расположены в памяти друг за другом. Поэтому возможен. прямой метод доступа к любому элементу;
- *Связанные,* у которых каждый элемент содержит информацию о других элементах этой же структуры. Прямой доступ к элементам связанных структур затруднен, а зачастую и невозможен.

Операции над структурами

- доступ к любому элементу;
- поиск индекса элемента по его значению;
- добавление нового элемента;
- изменение значений элемента с сохранением упорядоченности;
- удаление элемента;
- вставка нового элемента в произвольное место.

Для каждой из структур данных можно выделить как *удобные*, так и *неудобные* операции (т.е. такие операции, выполнение которых требует существенных затрат времени).

Упражнение: Какие операции являются удобными и неудобными для массивов?

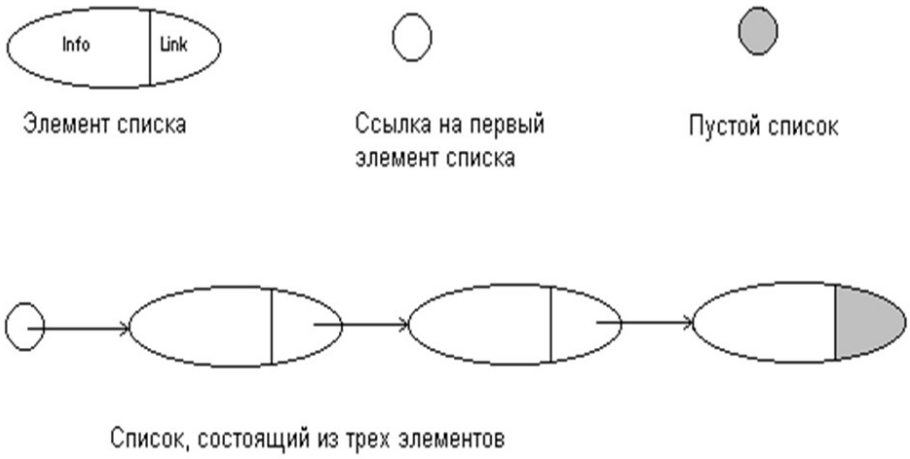
Абстрактные типы данныхЭто такие структуры, для которых нет соответствующих типов в языке программирования, но которые можно реализовать средствами языка.

Абстрактный тип данных "список"

Определение списков

Под списками (точнее, под ***однонаправлен-ными линейными списками***) будем понимать связанную структуру данных, каждый элемент которой содержит информацию о последующем элементе списка – т.н. ***ссылку*** на последующий элемент. Последний элемент списка содержит ***пустую*** ссылку. Именно по значению пустой ссылки можно определить, что текущий элемент списка - последний. Кроме этого, список должен содержать специальные данные – ***ссылку на первый элемент***.

Структура линейного однонаправленного списка



Так как каждый элемент списка содержит, по меньшей мере, два поля, то для его представления удобно использовать запись, содержащую поле **`Info`** произвольного типа (информационная часть) и поле **`Next`**, в котором хранится ссылка на последующий элемент. Тип поля **`Next`** зависит от физической организации списка.

Кроме того, в состав списка должна входить переменная **`First`**, которая хранит информацию о первом элементе списка.

Для списка удобными являются следующие операции:

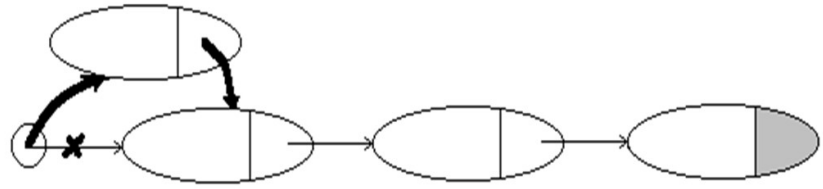
- вставка нового элемента в произвольное место списка (особенно – в

начало списка);

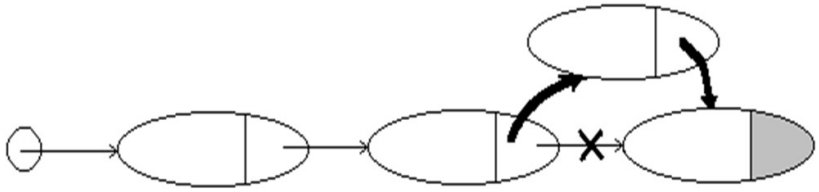
- удаление элемента из списка.

Обе эти операции требуют перестройки одной-двух ссылок.

Добавление нового элемента в список (схема)



Вставка нового элемента в начало списка



Вставка нового элемента в середину списка

Удаление элемента из списка (схема)

Удаление первого элемента списка

Удаление элемента из середины списка

Удаление последнего элемента списка

Для доступа к произвольному элементу списка необходимо пройти всю цепочку предшествующих элементов.

Поиск нужного элемента в отсортиро-ванном списке с использованием дихото-мии невозможен.

При удалении элемента из списка в статической памяти он не удаляется из

памяти, просто доступ к нему невозможен из-за того, что на удаленный элемент нет ссылок. Для физического удаления элемента списка из памяти следует выполнить т.н. процедуру ***сборки мусора***, которая предполагает освобождение неиспользуемых областей памяти. Алгоритмы сборки мусора сильно зависят от физической организации списка.

Реализация списков Наиболее часто используются следующие способы организации списков:

- ***с использованием динамической памяти;***
- ***с использованием массивов.***

Первый способ предполагает, что для каждого нового элемента списка выделяется участок динамической памяти.

После удаления элемента из списка можно освободить занятую этим элементом память, освобождая себя от дополнительных хлопот по сборке мусора.

Описание списка (реализация через динамическую память)

```
struct ListItem {
     int Info;
     ListItem* Next;
};
ListItem* First;
```

Теперь для задания пустого списка достаточно написать:

```
First = NULL;
```

Обработка списка

1. Добавление нового элемента в список

А) в начало списка

```
ListItem* P = new ListItem;
//заполнение поля P->Info
P->Next = First;
First = P;
```

Алгоритм:

1. Создать новый элемент типа список
5. Инициализировать его информационное поле.
6. Его ссылке присвоить значение из указателя на начало списка.
8. Изменить указатель на начало списка, занести в него адрес вставленного элемента.

В) После элемента, адрес которого находится в указателе Q

```
ListItem* P = new ListItem;
//заполнение поля P->Info
P->Next = Q->Next;
Q->Next = P;
```

Алгоритм:

1. Создать новый элемент типа список
5. Инициализировать его информационное поле.
6. Его ссылке присвоить значение из элемента списка, адрес которого находится в указателе Q.
8. Изменить ссылку в элементе, адрес которого находится в указателе Q, занести туда адрес вставленного элемента.

2. Удаление элемента из списка

А) из начала списка

```
if (First == NULL)
   throw "List is already empty!";
ListItem* P = First;
First = First->Next;
delete P;
```

В) После элемента, адрес которого находится в указателе Q

```
ListItem* P = Q->Next;
if (P == NULL)
   throw "Nothing to delete!";
Q->Next = P->Next;
delete P;
```

При удалении всего списка необходимо вначале уничтожить каждый элемент списка, а уже затем очищать значение указателя First. Заметим, что перед удалением элемента списка ссылка на него должна быть скопирована!

3. Просмотр всех элементов списка

```
ListItem* P = First;
while (P != NULL)
```

```
{
   // выполнить действия
   // над элементами P->Info
   P = P->Next;
}
```

Упр. ***Реализовать следующие функции обработки динамического списка:***

- Добавления нового элемента в начало списка;
- Добавления нового элемента после элемента, адрес которого известен;
- Поиска заданного элемента списка по его значению;
- Добавления нового элемента в список после заданного элемента;
- Распечатки значений элементов списка;
- Удаления элемента из начала списка;
- Удаления элемента после элемента, адрес которого известен;
- Удаления заданного элемента списка по его значению;
- Уничтожения списка.

Второй способ организации списка предполагает, что в памяти выделяется двумерный массив из N строк (N – максимальное число элементов, которые можно поместить в массив) и двух столбцов.

Организация списка в виде массива (пример)

Петров	7
Алексеев	8
Сидоров	-1
Борисов	2

Индекс начального элемента - 5

Сборка мусора при организации списка в виде массива

	4
	3
Петров	7
	0
	6
Алексеев	8
	9
Сидоров	-1
Борисов	2
	10
	11
	-1

Индекс начального элемента - 5
Индекс начального свободного элемента - 1

Каждая строка соответствует одному элементу списка, причем в первом столбце записывается содержимое элемента, а во втором – индекс строки со следующим элементом. Значение 0 соответствует последнему элементу списка.

Кроме того, переменная целого типа хранит номер строки с начальным элементом списка.

Упр. Нарисуйте вид списка после вставки в список элемента “Иванов”.

Реализация списка в виде массива

```
struct ListItem {
     char Info[50];
     int Next;
};

ListItem* List = new ListItem [N];

int First = -1, FirstFree = 0;
//со сборкой мусора
for (int i=0; i<N-1; i++)
   List[i].Next = i+1;
```

Другие виды списков

двунаправленные *или* ***двухсвязные*** списки. Элементы такого списка содержат ссылку не только на последующий, но и на предыдущий элемент. Сам список содержит, кроме того, ссылки на первый и последний элементы.

Такие списки используются, когда надо часто просматривать элементы списка в разных направлениях, например, при скроллинге различных таблиц.

Кроме того, двунаправленные списки позволяют восстановить значение одной ссылки при ее случайном разрушении и поэтому считаются более надежными структурами данных;

Кольцевые *или* ***циклические*** списки.

В кольцевых списках последний элемент содержит ссылку на первый. В них теряется смысл понятий “первого” и “последнего” элементов списка, вместо этого уместнее говорить о “текущем” элементе. Для таких списков удобной является операция перемещения текущего элемента вперед (а в случае двунаправленных списков – и назад).

Упражнения:

- Реализовать обработку двунаправленных линейных и кольцевых списков, представленных в динамической памяти.
- Реализовать обработку всех видов списков, представленных в виде массива записей.

Конец лекции