Лабораторная работа №1
по курсу «Структуры и алгоритмы обработки данных»

ПРЕДСТАВЛЕНИЕ ЛИНЕЙНЫХ СПИСКОВ

1. Основные сведения

1.1. Связанный список как структура данных

Связанные списки являются, наряду с массивами, одним из основных средств для построения сложных структур данных из более простых элементов. В массивах объединение элементов достигается путем их размещения в соседних, равных по размеру ячейках памяти. В связанных списках физическое размещение отдельных элементов не играет роли, а связь между элементами достигается путем явного указания для каждого элемента соседних с ним элементов.

Основным преимуществом связанных структур по сравнению с массивами является гибкость структуры. Операции добавления и удаления элементов структуры, а также изменения порядка элементов, выполняются путем изменения связей между элементами и не требуют перемещения самих элементов. В то же время связанные структуры не позволяют выполнить операцию быстрого доступа к элементу по его номеру, которая является основной операцией для массивов. Кроме того, связанные структуры требуют больше памяти для хранения каждого элемента (из-за необходимости хранить связи), но зато позволяют обойтись без предварительного резервирования памяти «по максимуму», как в случае массива. Таким образом, каждый из способов организации структур имеет свои достоинства и недостатки, что приводит к необходимости продуманного выбора структуры данных, наиболее эффективной для конкретной задачи.

В данной работе рассматриваются только ***линейные*** списки, в которых для каждого элемента (обычно называемого ***узлом*** списка) указан один (следующий) элемент либо два (следующий и предыдущий). Возможны и более сложные (нелинейные) связанные структуры, примерами которых являются деревья и графы.

1.2. Варианты представления линейных списков

1.2.1. Узел списка

Прежде чем говорить о представлении списка, следует разобраться с представлением отдельных элементов.

Узел списка можно представить с использованием, например, таких типов данных:

```
Type
  Element = ...;
  Link = ^Node;
  Node = record
    Data: Element;
    Next: Link;
    Prev: Link;
  end;
```

Здесь тип **Element** может означать практически что угодно: целое или вещественное число, строку символов, массив, запись, указатель и т.п. Тип **Node** представляет собой узел списка, а тип **Link** – связь с другим узлом. В данном случае это указатель на узел. Узел списка состоит из порции данных **Data** и связей со следующим и предыдущим

узлами **Next** и **Prev**. Список, содержащий две связи, называется ***двунаправленным*** или ***двусвязным***. Если присутствует только одна связь **Next**, список называется ***однонаправленным*** (***односвязным***).

Как теперь представить в программе сам список? Для этого достаточно иметь указатель на первый узел (***голову***) списка. Поэтому список как тип данных не отличается от связи с узлом.

```
Type
  List = Link;
Var
  List1, List2: List;
```

Иногда бывает полезно иметь также указатель на последний элемент списка (***хвост***). В этом случае правильным будет такое описание типа списка:

```
Type
  List = record
    Head: Link;  {Голова списка}
    Tail: Link;  {Хвост списка}
  end;
```

При перемещении по списку нужно иметь способ проверки на достижение конца списка. Чаще всего признаком конца служит специальное значение указателя **Next**, например, значение **nil**.

1.2.2. Списки «в куче» и списки «в массиве»

Если для связывания узлов используются указатели, то при этом обычно предполагают, что сами узлы размещаются в динамической памяти (куче). Память для хранения узла запрашивается у системы при создании этого узла и освобождается при его удалении. В языке Pascal для этого используются процедуры **New** и **Dispose** (или **GetMem** и **FreeMem**), в языке C – функции **malloc** и **free**, в C++ могут использоваться также операторы **new** и **delete**.

Другим возможным решением является размещение узлов списка в заранее выделенном массиве. Размер массива выбирается не меньше, чем максимально возможное число узлов. Каждая ячейка массива содержит узел списка и флаг, показывающий, занята или свободна данная ячейка. При создании нового узла программа должна найти любую свободную ячейку массива, записать в нее добавляемый узел и пометить ячейку как занятую. При удалении узла ячейка помечается как свободная. Естественно, что при размещении списка в массиве поля связи **Next** и **Prev** будут не указателями, а просто индексами ячеек массива. Специальным значением поля **Next** в конце списка может быть любое число, выходящее за пределы индексов массива (например, **-1**).

Размещение списков в массиве требует несколько меньше памяти на каждый узел и ускоряет выполнение операций по сравнению с размещением в куче (хотя поиск свободной ячейки может потребовать заметного времени). Недостатком размещения в массиве является необходимость знать заранее максимальный размер списка, в то время как память из кучи может запрашиваться по мере необходимости и ограничена только общими ресурсами памяти системы. Каждый из способов размещения имеет свои области применения. Например, списки в массивах широко используются в различных операционных системах для хранения внутренних системных данных, поскольку память для нужд системы обычно выделяется статически при загрузке.

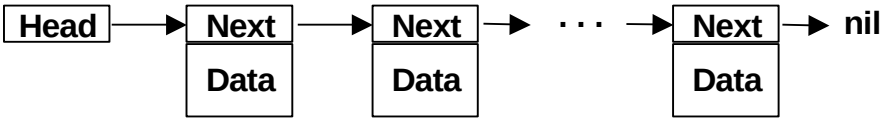
1.2.3. Варианты структуры списков

Выше уже отмечалось, что линейные списки могут быть односвязными и двусвязными. Преимущества односвязных списков заключаются, во-первых, в меньшем расходе памяти, и, во-вторых, в меньшем числе операций при изменениях списка. В то же время большим недостатком односвязных списков является невозможность сделать «шаг

назад», т.е. перейти от данного узла к предыдущему. Это затрудняет выполнение даже таких естественных операций, как удаление узла списка по заданному указателю или вставка нового узла перед указанным (хотя легко выполнить удаление или вставку *после* указанного узла). Двусвязные списки лишены этого недостатка.

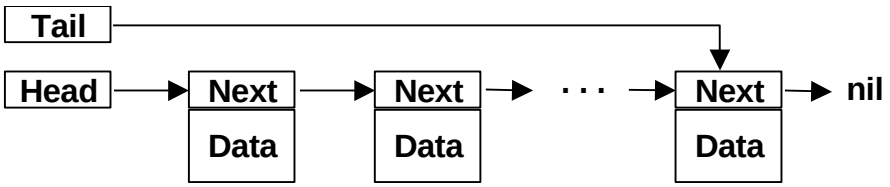
Как для односвязных, так и для двусвязных списков предложено несколько различных вариантов организации связей между узлами. Каждый из этих вариантов имеет свои достоинства и недостатки, и выбор наилучшего варианта определяется тем, какие операции будут чаще всего применяться к списку в конкретной задаче. Ниже описаны некоторые из вариантов организации списков.

1.2.3.1. Простой односвязный список



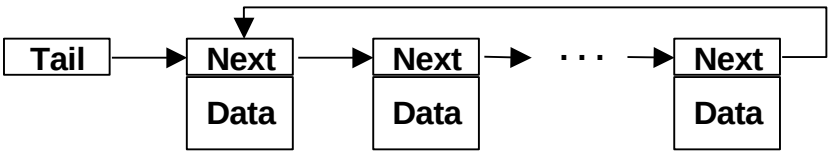
Самый обычный вариант линейного списка. Его недостаток в том, что для выполнения операций в хвосте списка приходится сначала в цикле переместиться от головы до хвоста.

1.2.3.2. Простой односвязный список с указателем на хвост



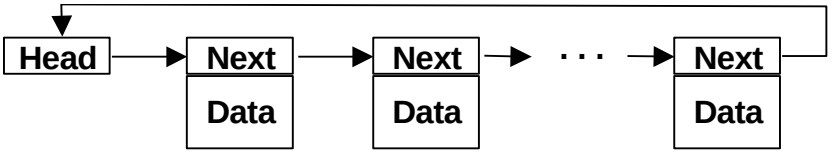
Это несложное усовершенствование позволяет упростить добавление элементов в хвост списка (но не удаление из хвоста). На практике подобные списки хороши для реализации «очередей», т.е. таких структур данных, у которых добавление данных происходит с одного конца, а удаление – с другого.

1.2.3.3. Циклический односвязный список с указателем на последний узел



В данном варианте списка удобно считать, что задан указатель не на первый, а на *последний* узел списка, а следующий за последним узел считать *первым*. Такой прием дает возможность легко получить доступ к обоим концам списка, без проблем выполняя операции вставки и удаления, как в голове, так и в хвосте списка. Однако требуется очень внимательно программировать операции над списком для случаев, когда в списке менее двух узлов.

1.2.3.4. Циклический односвязный список с зацикливанием «через указатель»

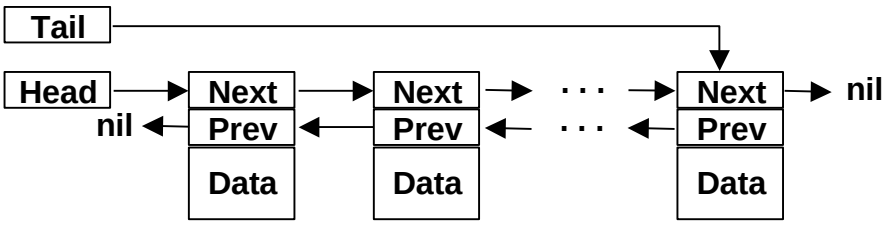


Такое несколько искусственное представление списка позволяет заметно упростить алгоритмы выполнения некоторых операций над списком, уменьшив количество ветвей

алгоритмов. Например, при удалении произвольного узла из простого линейного списка приходится проверять, не является ли удаляемый узел первым в списке, поскольку для первого узла требуется выполнить другие присваивания, чем для остальных узлов. Для данного же представления списка такого различия нет (убедитесь в этом самостоятельно). Отметим, что для пустого списка указатель **Head** не равен **nil**, а указывает сам на себя.

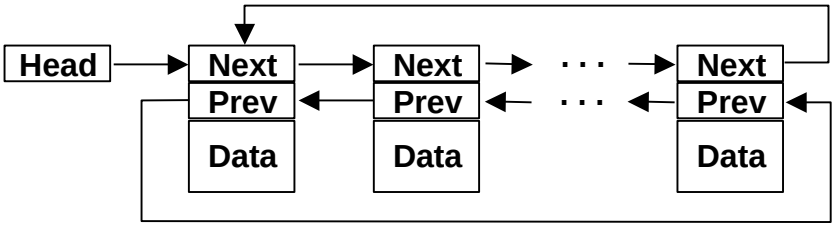
Некоторые сложности представляет согласование типов указателей. Например, для инициализации пустого списка не всегда удается просто написать «**Head := @Head**», это может быть синтаксической ошибкой (в зависимости от настроек компилятора). Более корректно использовать явное преобразование типов: «**Head := Link(@Head)**». При этом важно, чтобы поле **Next** было обязательно описано как *первое* поле записи **Node**.

1.2.3.5. Двусвязный линейный список с указателем на хвост



Двусвязный список с заданными указателями на первый и последний узлы снимает проблему перемещения по списку в обе стороны и тем упрощает операции вставки и удаления в любом месте списка. Ценой за это является необходимость хранить в каждом узле дополнительный указатель на предыдущий элемент и корректировать эти указатели при изменениях списка. Для пустого списка оба указателя **Head** и **Tail** должны быть равны **nil**.

1.2.3.6. Двусвязный циклический список



Более популярный вариант двусвязного списка. Отсутствие указателя на хвост списка компенсируется возможностью легко перейти от первого узла к последнему по указателю **Prev**.

1.3. Пример реализации операции со списком

В качестве примера того, как реализуется одна и та же операция для разных вариантов представления линейного списка, рассмотрим операцию добавления нового узла в хвост списка.

Будем рассматривать список, размещенный в динамической памяти. Тогда для всех вариантов списка следует, прежде всего, создать новый узел и заполнить его поле данных:

```
var p: Link;
...
p := New(Link);
p^.Data := ...;
```

Дальнейшие действия будут различны для разных представлений списка **L**. Чтобы разобраться в примерах, лучше всего нарисовать список карандашом и отслеживать выполняемые присваивания, изменяя направление соответствующих стрелок-указателей. Случай пустого списка следует рассмотреть отдельно.

1) Простой односвязный список:

```
p^.Next := nil;
if L = nil then
  L := p
else begin
  q := L;
  while q^.Next <> nil do
    q := q^.Next; {Вперед, к хвосту!}
  {Теперь q указывает на последний узел}
  q^.Next := p;
  p^.Next := nil;
end;
```

2) Простой односвязный список с указателем на хвост (**LTail**):

```
p^.Next := nil;
if L = nil then begin
  L := p;
  LTail := L;
end
else begin
  LTail^.Next := p;
  LTail := p;
end;
```

3) Циклический односвязный список с указателем на последний узел:

```
if L = nil then
  p^.Next := p
else begin
  {L указывает на последний, а не на первый узел!}
  p^.Next := L^.Next;
  L^.Next := p;
end;
L := p;
```

4) Циклический односвязный список с зацикливанием «через указатель»:

```
q := L;
while q^.Next <> Link(@L) do
  q := q^.Next;
{Теперь q указывает на последний узел}
p^.Next := q^.Next;
q^.Next := p;
{Случай пустого списка не отличается от общего случая}
```

5) Двусвязный линейный список с указателем на хвост (**LTail**):

```
if L = nil then
  L := p
else
  LTail^.Next := p;
P^.Next := nil;
p^.Prev := LTail;
LTail := p;
```

6) Двусвязный циклический список:

```
if L = nil then begin
  p^.Next := p;
  p^.Prev := p;
```

```
  L := p;
end
else begin
  p^.Next := L;
  p^.Prev := L^.Prev;
  L^.Prev^.Next := p;
  L^.Prev := p;
end;
```

2. Порядок выполнения работы

Выполнение лабораторной работы заключается в разработке и отладке программы работы со списками согласно полученному варианту задания.

Для выполнения работы можно использовать любой язык программирования, содержащий необходимые средства. Продвинутый интерфейс программы допускается, но совершенно не обязателен.

Отчет о лабораторной работе оформляется на бумаге в печатном или рукописном виде. На титульном листе указывается название работы и состав бригады. В отчете приводится формулировка задания, а также комментированный текст разработанной программы.

Отчет и программа представляются также в электронном виде.

В прилагаемом проекте **Laba1** содержится пример программы, выполняющей основные операции для одного из вариантов представления списков. Можно использовать текст примера как основу.

3. Варианты заданий

Для *всех* вариантов нужно реализовать меню со следующим минимальным набором операций со списком:

- инициализация пустого списка;
- уничтожение списка с освобождением памяти;
- добавление узла в голову списка;
- добавление узла в хвост списка;
- удаление узла из головы списка;
- удаление узла из хвоста списка;
- выдача текущего списка на экран.

Тип данных, хранящихся в узлах, выбирается по желанию студента. Метод размещения списка (в динамической памяти или в массиве) также может быть выбран произвольно.

Кроме того, в каждом варианте следует реализовать две дополнительные операции. Вариант представления списка и дополнительные операции выбираются согласно номеру варианта.

1) Циклический односвязный список с указателем на последний узел. Дополнительные операции: а) инвертировать список (изменить порядок узлов на обратный); б) перенести (не копируя) один список в хвост второго.
2) Циклический односвязный список с зацикливанием «через указатель». Дополнительные операции: а) перенести (не копируя) все нечетные по порядку узлы в отдельный список; б) добавить новый узел после узла с заданным значением данных.
3) Двусвязный линейный список. Дополнительные операции: а) добавить новый узел в указанную позицию; б) поменять местами первый и последний узлы (требуется поменять именно узлы, а не их значения).
4) Двусвязный циклический список. Дополнительные операции: а) добавить (скопировать) один список в хвост второго; б) перенести (не копируя) все четные по порядку узлы в отдельный список.

5) Циклический односвязный список с указателем на последний узел. Дополнительные операции: а) поменять местами первый и второй (если он есть) узлы (требуется поменять именно узлы, а не их значения); б) удалить узел с указанным порядковым номером.
6) Циклический односвязный список с зацикливанием «через голову». Дополнительные операции: а) добавить новый узел в указанную позицию; б) удалить узлы с заданным значением данных.
7) Двусвязный линейный список. Дополнительные операции: а) удалить узлы с заданным значением данных; б) инвертировать список (изменить порядок узлов на обратный).
8) Двусвязный циклический список. Дополнительные операции: а) добавить новый узел перед узлом с заданным значением данных; б) удалить узел с указанным порядковым номером.