V.05.

Общий обзор занятия.

Общий обзор формулируется очень просто: продолжаем строить структуры данных, как можно более адекватные выбранному алгоритму решения задачи. И снова обязательно подчёркиваем именно эту линию:

```
задача -> метод (алгоритм) решения -> необходимые способы доступа к данным -> кон
струируем соответствующую реализацию структуры данных
```

Так что опять

- Рассмотрим пару задач. Одна из них - задача о максимуме в плавающем окне - уже была, но мы придумаем для неё новый, более эффективный алгоритм, и, сответственно, построим другую, более соответствующую структуру данных. Этот момент очень показателен.
- Попридумываем для них решения,
- а затем будем искать способы представления/хранения данных, по возможности максимально адекватные тем способам доступа к данным, которые требует придуманный способ решения,
    - и конструируем реализацию придуманного абстрактного типа данных, стараясь подобрать максимально адекватные техники хранения.

Начнём, как уже сказано, с задачи о максимумах в плавающем окне, но новое решение потребует другой структуры хранения данных, и это - дек (двусторонняя очередь), которая потребует двунаправленного (двусвязного) списка. Решение это не то чтобы уж совсем страшное, но такое, неочевидное. Во всяком случае, его более высокая эффективность не совсем очевидна. Т.е. понятно, что оно эффективнее, но не сразу видно, что это ускорение даёт качественно другую скорость. Ну, и, соответственно, вторую задачу возьмём алгоритмически попроще, но с некоторыми небольшими вопросиками по реализации - задача Иосифа и структура данных кольцо. Да, и окажется, что тут вполне хватит однонаправленного хранения.

Как и большинство линейных структур (линейных в смысле структур, где данные выстроены в линию, не разветвляются, где узлы в любой момент можно просто пронумеровать по порядку) эти структуры можно реализовать на слайсе, но уже в случае с задачей Иосифа, где надо вычёркивать узлы из середины списка, а не с краёв, связные списки вполне могут ускорить обработку. Но решения на слайсах тоже рассмотрим, хоят и без особого обсуждения - дети вполне уже сами могут разобраться, пусть и разбираются.

Ну, и, повторюсь в очередной раз, каждый раз обращаем внимание явно на связь АТД <-> реализация, т.е. набор операций, набор способов доступа к данным <-> конкретная реализация имеющимися средствами конкретного средства программирования (языка Go в нашем случае).

Так что в этот раз опять у нас есть и принципиальный момент - конструирование АТД, и технический момент - реализация. И опять, пока дети ещё совсем не привыкли, пока у них ещё нет уверенного навыка работы со связными структурами, техническому моменту надо уделять достаточно много внимания, особенно на практике. Пожалуй даже больше, чем принципиальному. Техника сейчас крайне важна. А достигается техника, конечно, упражнениями. Не скупитесь :) Много упражнений дают нам просто связные списки. Какие-то упражнения уже были в прошлом занятии (по крайней мере в плане), какие-то я приведу в этом тексте, но простор для фантазии здесь широчайший.

Лекция

Задача о максимуме на плавающем окне - дек.

Задача, которой мы закончили в прошлый раз. И сегодня рассмотрим обещанное более эффективное решение, использующее дек, который удачно требует двусвязного списка.

Повторю формулировку задачи: есть ряд из N чисел, и есть окно на K элементов, K < N. Окно двигается слева направо от начала ряда чисел к концу, и каждый раз нас интересует максимальное значение, видное в данный момент в окне. Понятно, что на выходе мы ожидаем N-K+1 максимумов. Например, если мы имеем ряд из 7 чисел

```
4 7 5 3 6 11 9,
```

а K = 3, то на выходе мы получаем

```
7 7 6 11 11
```

Обсуждение решения (алгоритма) и построение алгоритма

Ну, что, начинаем двигать окно. Смотрим на то, как изменяется содержимое окна. Первые K шагов окно наезжает на начало ряда чисел и увеличивается в размерах. При этом максимум вычислять пока не надо (кроме последнего раза, но перенесём это действие на следующий этап). Вот что происходит:

```
1-й шаг. Окно содержит: 4
2-й шаг. Окно содержит: 4 7
3-й шаг. Окно содержит: 4 7 5
```

И теперь начинается второй этап - движение заполненого окна. Первым делом вычисляем максимум в полученном окне, получаем 7, а затем продвигаем окно. При этом число, стоящее на левом краю, - это 4 - уходит из окна, а на правый край добавляется следующее число из данного ряда чисел - это 3.

```
4-й шаг. Окно содержит: 7 5 3
```

Продолжаем. Находим текущий максимум в окне - это 7. И теперь начинаем двигать наше окно. Выполняем пятый шаг. Следующее число в списке - число 6. Поставим его на правый край, получим ряд `7 5 3 6`. Стоп! Даайте на минутку задумаемся. А нужно ли нам хранить число 3? Может оно нам ещё понадобиться? Кажется, что нет. Оно меньше 6 и оно покинет окно раньше 6, так что тройка никогда не сможет стать максимумом в окне. Так давайте выкинем его. Остаётся `7 5 6`. Так и пятёрка попадает под трамвай (в смысле под число 6) точно также, как и тройка. Так что и 5 нам тоже хранить не нужно. А вот об семёрку вновь пришедщая шестёрка обламывается. Ну, и ладно. выкидываем 5, ставим на правый край 6, получаем `7 6`. Так, но теперь бы надо нам семёрку убрать. А чего это мы её будем убирать? Убирать число из окна надо тогда, когда в окне окажется больше K чисел, в нашем примере K=3, а чисел мы держим только два: 7 и 6. Да, но мы же числа выкидывали, и 7 надо выкинуть из-за того, что оно в исходном ряду чисел стояло на втором месте, т.е. из-за того, что его номер достаточно маленький. Так. Значит получается, что, кроме самих чисел, нам надо хранить номера их позиций в исходном ряду. Что-то пока плюсов особо не видно, а хранить что-то дополнительно уже надо. Но ладно, давайте попробуем. Продолжаем выполнять пятый шаг. Мы сейчас храним два числа: `7[2], 6[5]`. В квадратных скобках - номер пзиции числа в исходном ряду чисел, в том ряду, по которому движется окно. Шаг пятый, значит семёрку, которая находится на втором месте, надо убирать. Убираем. В итоге, после пятого шага, мы храним одно единственное число - `6[5]`. Найти максимум легко :) Нашли, вывели.

Переходим к шестому шагу. Приходит число 11[6]. 11 > 6, шестёрку выкидываем, больше выкидывать нечего, но 11[6] всё-таки оставим.

Ну, и последний шаг - шаг номер 7. На седьмом месте в ряду чисел стоит число 9. 9 < 11, так что просто дабавляем 9[7] на правый край. А потом ищем максимум их тех чисел, коортые мы храним. Это `11[6], 9[7]`. И максимум нахоим и выводим.

А, кстати, а чего его искать, максимум. Ведь так, как мы теперь храним числа, максимум может находится только на самом левом месте.

О, это уже интересно - мало того, что мы теперь храним меньше чисел, так ещё и не надо вдоль них бегать в поисках максимального. А это реально серьёзное ускорение.

Давайте сформулируем вместе, что же у нас получилось:

- добавляем новое число X в правый край окна; для этого:
    - если самое правое число из тех, которые мы храним, меньше X, то мы это самое правое число выкидываем
    - повторяем эту операцию до тех пор, пока самое правое хранимое число не окажется больше X либо хранимых чисел не останется
    - ставим число X справа от оставшихся чисел
- смотрим номер самого левого хранимого числа, если он достаточно маленький, если это число пора убирать из окна, то убираем его; ясно, что на каждом шаге убирать надо не более одного числа
- выводим самое левое их хранимых чисел - это максимальное из чисел в окне.

Остаётся ещё пара мелочей.
Первая. А что если хранимое число равно X? Понятно, что его тоже можно выкинуть - новый X по величине такой же, а находится правее.
Вторая. Можем переставить местами первое и второе действия: сначала пытаться выкинуть левое число, а потом добавлять новое число в правый край.

Сформулируем ещё раз, уже окончательно, что же у нас получилось.
И давайте уже перестанем оиспользовать слова “хранимые числа” - ведь это и есть окно, точнее, те числа из окна, которые нам интересны. Ну так и давайте называть это окном или содержимым окна.

1. Выкидываем, если надо самое левое число в окне:
    - смотрим номер самого левого числа в окне, если он достаточно маленький, если это число пора убирать из окна, то убираем его; ясно, что на каждом шаге убирать надо не более одного числа
2. Добавляем новое число X в правый край окна; для этого:
    - если самое правое число в окне меньше или равно X, то мы это самое правое число выкидываем
    - повторяем эту операцию до тех пор, пока в окне есть числа, и самое правое из них не окажется больше X
    - ставим число X справа от оставшихся чисел
3. Выводим максимальный элемент окна:
    - максимальный элемент окна - это самое левое из чисел окна, ведь числа в окне (повторю на всякий случай, что речь идёт только о тех числа , которые мы храним) убывают слева направо

Всё.

Какие операции, какие доступы к данным нам потребуются?

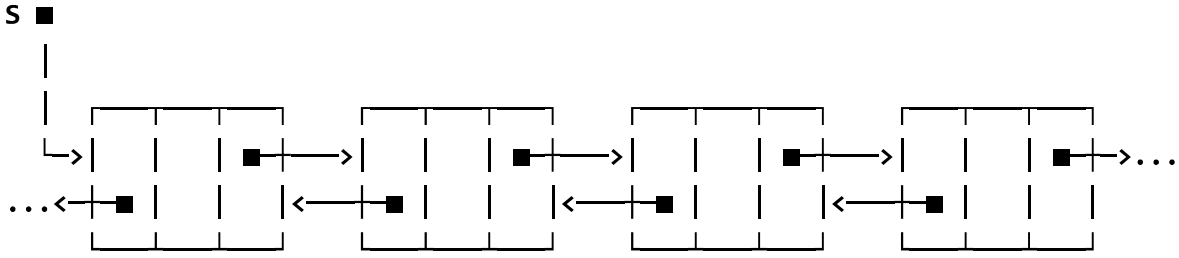
```
- прочитать элемент на левом краю
- удалить элемент на левом краю
- прочитать элемент на правом краю
- удалить элемент на правом краю
- добавить элемент на правый край
```

Этакая очерель, только с приходом-уходм с обоих концов. Тяни-толкай такой. В данной задаче, правда, нет необходимости добавлять элементы с левого края, но это вот такая особенность именно этой задачи, использовать её всё равно не получается (хотя, может кто-нибудь когда-нибудь придумает что-нибудь очень специфическое именно для такого набора доступов).

А пока применим хорошо известную структуру данных - дек, они же двустороння очередь. Собственно, название дек, по-английски deque - сокращение от названия double-ended queue

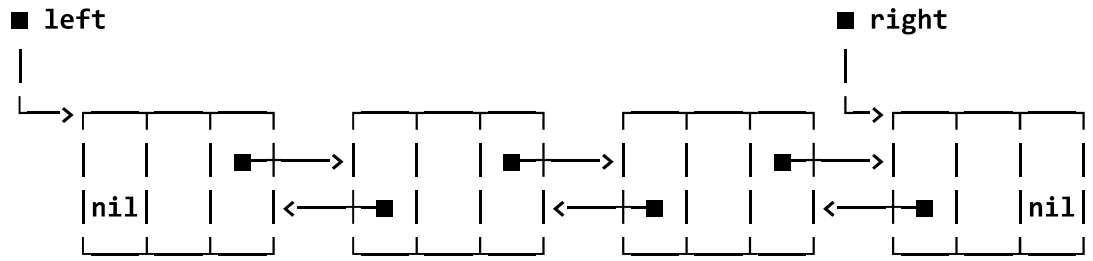
Обсуждение реализации

Разумеется, дек, как и очередь, можно запросто реализовать на слайсе. Разве что, как и с очередью тут есть необходимость удалять элемент из начала слайса, а не с конца, а это может быть времязатратно. Но мы обсудим реализацию дека на связной структуре, а уж написать два решения (со слайсом и со связной структурой), побенчмарчить и сравнить их - это святое.
И вот тут-то начинается интересно… Нам необходимо удалять и самый левый элемент, и самый правый элемент. А удаление элемента с края требует перехода на следующий с этого края (бывшего второго с этого края) элемент. Но это значит, что тот элемент, котрый мы удаляем с правого края должен быть связан с элементом, слева от себя, а тот, который удаляем с левого края, - с лементом справа от себя. Но ведь люьой элемент может пройти путь с левогь края на првый или наоборот. Получается, что каждый элемент должен быть связан с обоими соседями? Так таки да! И ничего в этом страшного нет. Да, вместе с каждым элементом будем хранить два указателя - и на соседа слева, и на соседа справа. Вот как на рисунке:



Такая конструкция называется *двунаправленный связный список*, или, более кратко, *двусвязный список*

Задумаемся немножко и убедимся, что двусвязный список обеспечивает всё, что нужно для эффективой (и весьма) реализации дека. Ну, разве что, поскольку смотреть/удалять надо элементы с обоих концов, то мы должны держать этот двусвязный список за оба конца. Я не знаю, где у Тянитолкая хвост, поэтому назову края двусвязного списка просто левый и правый. Итого получаем такое хранение - классический двусвязный список:



Писать реализации просто с функциями я в это раз уже не буду, приведу две реализации с методами типа `deque` : одна с реализацией дека на слайсе (numbers_03.go), другая - на двусвяхном списке (numbers_04.go). 03 и 04 - это потому, что эта задача уже была реализована два раза на прошлом занятии.

Программа numbers_03.go.

```
package main

import     (
    "fmt"
    "errors"
    "os"
    "math"
)

type (
    lmnt struct {
        val int
        pos int
    }
    deque []lmnt
)

func NewDeque() deque  {
    return make([]lmnt, 0, 0)
}

func (dq *deque) PushLeft (w lmnt) {
```

```
    *dq = append([]lmnt{w}, (*dq)...)
}

func (dq *deque) PushRight (w lmnt) {
    *dq = append(*dq, w)
}

func (dq deque) GetLeft() lmnt {
    if dq.Empty()  {
        return lmnt{math.MinInt64, -1}
    } else {
        return dq[0]
    }
}

func (dq deque) GetRight() lmnt {
    if dq.Empty()  {
        return lmnt{math.MinInt64, -1}
    } else {
        return dq[len(dq)-1]
    }
}

func (dq *deque) RemoveLeft () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    }  else  {
        *dq = (*dq)[1:]
        return nil
    }
}

func (dq *deque) RemoveRight () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    } else {
        *dq = (*dq)[:len(*dq)-1]
        return nil
    }
}

func (dq deque) Empty() bool  {
    return len(dq) == 0
}
```

```
func main() {
    f, err := os.Open("numbers.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    win:= NewDeque()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }

        for !win.Empty() &&  win.GetRight().val <= x {
            win.RemoveRight()
        }
        win.PushRight(lmnt{x, i})
    }
    fmt.Println(win.GetLeft().val)

    for i:= k; ; i++ {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }

        if win.GetLeft().pos == i-k {
            win.RemoveLeft()
        }

        for !win.Empty() &&  win.GetRight().val <= x {
            win.RemoveRight()
        }
        win.PushRight(lmnt{x, i})

        fmt.Println(win.GetLeft().val)
    }
}
```

Программа numbers_04.go.

```
package main
```

```
import     (
    "fmt"
    "errors"
    "os"
    "math"
)

type (
    data struct {
        val int
        pos int
    }
    lmnt struct {
        d data
        l *lmnt
        r *lmnt
    }
    deque struct {
        left *lmnt
        right *lmnt
    }
)

func NewDeque() deque  {
    return deque{nil, nil}
}

func (dq *deque) PushLeft (d data) {
    if (*dq).Empty()  {
        (*dq).left = &lmnt{d, nil, nil}
        (*dq).right = (*dq).left
    } else {
        (*(*dq).left).l = &lmnt{d, nil, (*dq).left}
        (*dq).left = (*(*dq).left).l
    }
}

func (dq *deque) PushRight (d data) {
    if (*dq).Empty()  {
        (*dq).left = &lmnt{d, nil, nil}
        (*dq).right = (*dq).left
    } else {
        (*(*dq).right).r = &lmnt{d, (*dq).right, nil}
        (*dq).right = (*(*dq).right).r
```

```
    }
}

func (dq deque) GetLeft() data {
    if dq.Empty()  {
        return data{math.MinInt64, -1}
    } else {
        return (*dq.left).d
    }
}

func (dq deque) GetRight() data {
    if dq.Empty()  {
        return data{math.MinInt64, -1}
    } else {
        return (*dq.right).d
    }
}

func (dq *deque) RemoveLeft () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    }
    if (*dq).left == (*dq).right  {
        (*dq).left, (*dq).right = nil, nil
    } else {
        (*dq).left = (*(*dq).left).r
    }
    return nil
}

func (dq *deque) RemoveRight () error {
    if (*dq).Empty()  {
        return errors.New("Attempt to remove from empty deque")
    }
    if (*dq).left == (*dq).right  {
        (*dq).left, (*dq).right = nil, nil
    } else {
        (*dq).right = (*(*dq).right).l
    }
    return nil
}

func (dq deque) Empty() bool  {
```

```
    return dq.left==nil
}

func main() {
    f, err := os.Open("numbers.dat")
    if err != nil {
        return
    }
    defer f.Close()
    var k int
    _, err = fmt.Fscanf(f, "%d\n", &k)
    if err != nil  { return }
    var x int
    win:= NewDeque()
    for i:= 0; i < k; i++  {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err !=nil  { break }

        for !win.Empty() &&  win.GetRight().val <= x {
            win.RemoveRight()
        }
        win.PushRight(data{x, i})
    }
    fmt.Println(win.GetLeft().val)

    for i:= k; ; i++ {
        _, err := fmt.Fscanf(f, "%d\n", &x)
        if err != nil  { break }

        if win.GetLeft().pos == i-k {
            win.RemoveLeft()
        }

        for !win.Empty() &&  win.GetRight().val <= x {
            win.RemoveRight()
        }
        win.PushRight(data{x, i})

        fmt.Println(win.GetLeft().val)
    }
}
```

Анализ эффективности нового решения задачи о

максимумах в плавающем окне - мелкий шрифт.

Понятно, что новое решение эффективнее старого, уже хотя бы потому, что нам не надо каждый раз пробегать по всему окну в поисках максимума, не говоря уж о том, что само окно (его хранимая часть) уменьшилось в размере. Но насколько эффективнее - вопрос. Всё-таки, кажется мне, что тут образуется некоторый выброс за рамки наших интересов и возможностей, но не изложить такую красоту невозможно. Так что пусть оно будет, но мелким шрифтом (типа, если всё хорошо и очень хочется изложить - то можно, хотя бы некоторым).

Начало мелкого шрифта

При реализации алгоритма мы выполняем операции трёх типов:

1. Выкидываем, если надо самое левое число в окне:

```
- смотрим номер самого левого числа в окне, если он достаточно маленький, если эт
о число пора убирать из окна, то убираем его; ясно, что на каждом шаге убирать на
до не более одного числа
```

Всего выполняется n таких операций. Точнее n-k+1, но не будем мелочиться.

2. Добавляем новое число X в правый край окна; для этого:

```
- если самое правое число в окне меньше или равно X, то мы это самое правое число
 выкидываем
- повторяем эту операцию до тех пор, пока в окне есть числа, и самое правое из ни
х не окажется больше X
- ставим число X справа от оставшихся чисел
```

Всего операций удаления не может быть больше, чем n. Ведь каждое число после того, как мы его удалим, больше не участвует в сюжете вообще, оно уходит навсегда. Так что каждое число можно выкинуть максимум один раз, а чисел всего n. Ну, и каждое из n чисел мы ровно один раз ставим на правый край, таких операций, понятное дело, ровно n.

3. Выводим максимальный элемент окна:

```
- максимальный элемент окна  - это самое левое из чисел окна, ведь числа в окне (
повторю на всякий случай, что речь идёт только о тех числа , которые мы храним) у
бывают слева направо
```

Взять самое левое число - одна операция, всего их n штук. Опять-таки, их, конечно, n-k+1, но не будем мелочиться.

Итого получаем не более `4*n` операций - от размера окна не зависит. И это ура!

Конец мелкого шрифта

Заключительное замечание

Легко видеть, что полученное решение прекрасно работает и в более общем случае, когда края окна перемещаются независимо: можно сдвигать вправо правый край окна, можно сдвигать вправо левый край окна, при каждом движении любого края требуется выводить максимум в получающемся окне.

Задача Иосифа

Второй пример – алгоритмически попроще, чисто задача на реализацию – задача Иосифа, она же Josephus problem.

Описание задачи

О присхождении названия и за формулировкой задачи – сюда: http://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%98%D0%BE%D1%81%D0%B8%D1%84%D0%B0_%D0%A4%D0%BB%D0%B0%D0%B2%D0%B8%D1%8F.
Хотя… Там немного, так что приведу цитату:
Задача Иосифа Флавия или считалка Джозефуса — известная математическая задача с историческим подтекстом. Задача в своей основе имеет легенду. Отряд из 41-го сикария, защищавший галилейскую крепость Массада, не пожелал сдаваться в плен блокировавшим его превосходящим силам римлян. Сикарии стали в круг и договорились, что каждые два воина будут убивать третьего, пока не погибнут все. Самоубийство — тяжкий грех, но тот, кто в конце концов останется последним, должен будет его совершить. Иосиф Флавий, командовавший этим отрядом, якобы быстро рассчитал, где нужно стать ему и его другу, чтобы остаться последними. Но не для того, чтобы убить друг-друга, а чтобы сдать крепость римлянам. В современной формулировке задачи участвует N воинов и убивают каждого M-го. Требуется определить номер k начальной позиции воина, который должен будет остаться последним.
Или по-английски http://en.wikipedia.org/wiki/Josephus_problem :
There are people standing in a circle waiting to be executed. After the first person is executed, certain number of people are skipped and one person is executed. Then again, people are skipped and a person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom.

Обсуждение задачи

Начнём, как и писали выше, с обсуждения задачи, с методов решения. Каких-то особых хитростей в решении, как это случилось с предыдущей задачей, в данном случае не видно, будем решать задачу, просто симулируя процесс вычеркивания воинов из списка.

Ну, а раз ничего такого особо хитрого мы делать не планируем (кашляну в сторону: на самом деле тут можно сделать и кое-что хитрого, весьма хитрого и довольно эффективного, но это совсем другая история…), раз уж всё равно мы последовательно ищем и вычёркиваем воинов, то будем выводить номера всех воинов в том порядке, в каком их убивают. Например, если N=10, а M=4, то убиваем воинов в таком порядке:

```
    4, 8, 2, 7, 3, 10, 9, 1, 6 и 5 .
```

Вот такую задачу мы и будем решать.

Какие действия нам надо исполнять?

```
1. перемещаться на M позиций по кругу
2. вычеркивать элемент
```

Естественно, есть сразу два конкурирующих подхода: реализовывать симуляцию процесса, храня данные в слайсе либо храня данные в какой-то связной структуре, что-то типа связного списка.

Попробуем взглянуть на эти подходы совсем с общих позиций, вообще отстраняясь пока от деталей. Перемещаться надо на M позиций, причём это M может быть довольно большим. Списковая структура не позволяет нам перемещаться по списку быстро - только пошагово, так что от списковой структуры не приходится ожидать эффективной реализации операций первого вила - перемещения на M позиций. А вот слайс, в силу того, что элементы в нём проиндексированы, пронумерованы, весьма вероятно позволит быстро-быстро перемещаться на сколько угодно позиций, вычисляя конечную точку перемещения арифметическими методами, а не исполняя M шагов. Хотя, конечно, слайс есть отрезок, а не кольцо, это немного настораживает, кто его знает, это может и как-то помешать. Забегая вперёд, скажу, что эти опасения не оправдаются. Однако, с другой стороны, есть и вторая операция - вычёркивание элемента. И здесь связные структутры показывают себя превосходнейшим образом - для вычеркивания элемента из списка достаточно нескольких операций, зачастую достаточно даже одной-двух. А вот слайсы, как мы помним, с вычёркиванием элементов тормозят запросто. Так что однозначно говрить, что тот или иной способ лучше, мы пока не можем. Ну, разве что скажем, что если M велико, то вероятно лучше хранить данные в слайсе, а если M небольшое, то предпочтительнее могут оказаться списки.

Так что попробуем реализовать оба подхода. Было бы неплохо поисследовать, при каких соотношениях N и M, какой из подходов работает быстрее.

Задача Иосифа. Реализация на слайсе.

Этот подход тоже допускает различные варианты, но остановимся на одном: храним в слайсе имена воинов – их номера. Да, собственно, какая разница, как их зовут, нам нужно их однозначно идентифицировать любым способом, так что здесь речь идёт об уникальном идентификаторе. Трудность, связанная с “незацикленностью” слайса решается без проблем. Легко видеть, что если мы стоим на позиции номер k, то сделав M шагов вперёд по кругу, мы окажемся на позиции p = (k+M)%len, где len - это длина слайса, количество элементов (имён) в нём в данный момент. И тогда реализация не вызывает проблем.

Программа josephus_01.go.

```
package main

import     (
    "fmt"
    "errors"
    "math"
)

type (
    lmnt int
    ring struct {
        row []lmnt
        currentPos int
    }
)

func NewRing(len int) ring {
    var r ring
    r.row = make([]lmnt, len, len)
    r.currentPos = 0
    return r
}

func (r ring) Empty() bool  {
    return r.Length() == 0
}
```

```
func (r ring) Length() int  {
    return len(r.row)
}

func (r ring) GetCurrent() lmnt {
    if r.currentPos >= r.Length() || r.currentPos < 0 {
        return math.MinInt64
    } else {
        return r.row[r.currentPos]
    }
}

func (r *ring) RemoveCurrent () error {
    if r.currentPos >= r.Length() || r.currentPos < 0 {
        return errors.New("Invalid removing")
    }  else  {
        (*r).row = Delete((*r).row, (*r).currentPos)
        return nil
    }
}

func Delete(s []lmnt, i int) []lmnt {
    return s[:i+copy(s[i:], s[i+1:])]
}

func (r *ring) Forward (step int) error {
    if (*r).Empty() {
        return errors.New("Invalid step")
    }  else  {
        (*r).currentPos = ((*r).currentPos + step) % (*r).Length()
        return nil
    }
}

func main() {
    var N, M int
    fmt.Print("Enter N: ")
    fmt.Scanf("%d\n", &N)
    fmt.Print("Enter M: ")
    fmt.Scanf("%d\n", &M)

    r:= NewRing(N)
    for i, _ := range r.row  {
        r.row[i] = lmnt(i+1)
```

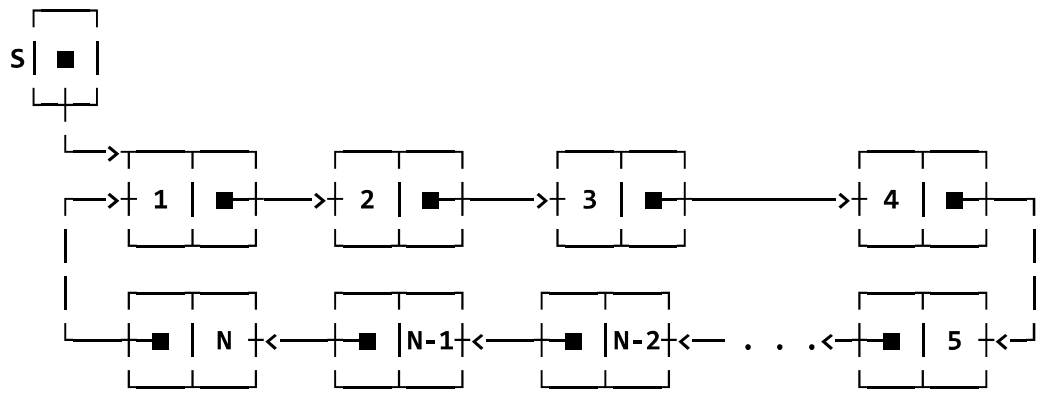
```
    }

    for i:= 0; i < N; i++ {
        r.Forward(M-1)
        fmt.Println(r.GetCurrent())
        r.RemoveCurrent()
    }
}
```

Программа понятная, написано прозрачно, у меня при критическом просмотре возник только один вопрос: почему прыгаем вперёд не на M шагов, а только на M-1? Ответ несложный - когда мы удаляем элемент (имя воина, его номер), мы автоматически перепрыгиваем на следующий, т.е. один шаг уже делаем. ТОчнее шговоря, не мы перепрыгиаваем, а следующий элемент (вместе со всеми остальными) переползает на одну позицию влево, подлезая тем самым под указатель текущей позиции. Разобрать программу, проётись поней, тем не менее, стОит, но проблем здесь не видно.

Задача Иосифа. Реализация на кольцевом списке (кольце).

Да, ключевой момент уже всплыл в заголовке - раз нам надо двигаться по кругу, т.е. вслед за последним воином стоит первый, так и давайте составим из них закольцованный список или связное кольцо. Каждый воин “видит” следующего, хранит при себе указатель на следующего, а последний, естественно, хранит указатель на первого.



А поскольку “у кольца нет ни начала, ни конца”, то держать его, это кольцо, можно за любое место, за любой элемент. На рисунке мы держим его за единицу, а в программе мы так строим кольцо, что держим его перед началом движения за число N. В самом деле, тогда наш первый шаг вперёд и отсчитает элемент 1 - тот, с которого начинается отсчёт.

main пишется довольно легко, все процессы ясны, но на некоторых моментах надо бы

остановиться. Ясно, что ключевое действие здесь - удаление элемента. И вот тут мы натыкаемся на то, что в односвязном кольце удалить текущий элемент тяжело (ну, можно, конечно, но для этого придётся пробежать полный круг), ведь нам надо при удалении элемента модифицировать предыдущий элементе - изменить у него указатель на следующий. Простые выходы:

- делать почти полный круг, чтобы остановиться на предыдущем
- держать при переходах кольцо за два места - за текущий элемент и за предыдущий
- сделать кольцо двусвязным, чтобы каждый элемент видел не только следующего, но и предыдущего
  критикуются легко. Тут тебе и снижение эффективности, и усложнение обработки, и некая неэлегантность - несоответствие выбранных средств задаче, и т.д. Однако первый вариант, несмотря на его самую уж тормознутость, посмотрев на второй вариант, подсказывает нам выход, выход простой и как раз элегантный - держать кольцо только за предыдущий элемент: удалить или рассмотреть следующий эдлемент не представляет труда, да и код усложняется микроскопически.

Пару слов надо бы сказать (а гораздо лучше порисовать) о создании кольца, о накачке его элементами. Сначала создаём одно элементное кольцо, содержащее номер N. А потом уже начинаем перед ним, не сдвигаясь с места вставлять номера от N-1 убывая до 1. И получается то, что надо - мы как бы отталкиваем от N номера, и чем больше номер, тем раньше его надо ставить, тем сильнее оттолкнуть. Но рисунок здесь объяснит всё лучше, чем слова.

Пробегание по кольцу тоже имеет пару забавных моментов, на которых сто́ит остановится: раз уж останавливаться надо перед вычеркиваемым элементом, то каждый раз надо делать не M прыжков, а только M-1. Если M дольшое, то можно ускорить программу – хранить количество оставшихся людей и делать не M-1 прыжок, а (M-1)% длина_кольца. О, хранить количество оставшихся людей, оно же длина кольца. Для этого у нас есть метод Length. Вопрос, как его реализовывать? Можно, конечно, бегать по кольцу полный круг и подсчитывать количество элементов. Но в данном случае выбрано более эффективное и, как по мне, гораздо более естественное решение - просто вместе с кольцом хранить его длину, изменяя длину при соответствующих изменениях - добавлениях и удалениях элементов. В итоге тип кольца стал struct'ом с двумя полями:

```
ring struct {
    current *lmnt
    len int
}
```

а реализация метода Length() стала выглядеть немножко идиотично формальной, но пусть будет метод. Давайте уже сейчас приучать детей не злоупотреблять, я бы даже

сказал, избегать прямого обращения к полям. В данном случае довод простой: передумаем - уберём в описании типа ring поле len, уберём из всех методов упоминание поля len (а оно упоминается минимально - именно там, где изменяется длина кольца - при удалениях и вставках) и перепишем метод Length(). И всё.

Вроде бы и всё. Нет, конечно, методы надо бы и разобрать хоть немного, хоть пробежаться по ним, но так, без фанатизма. Время для разговора о проектировании, о том, почему именно такой, а не иной набор методов мы выбираем, пока ещё не настало.

Вот этот код и реализует задачу Иосифа на связном кольце.
Программа josephus_02.go.

```
package main

import     (
    "fmt"
    "errors"
    "math"
)

type (
    lmnt struct {
        n int
        next *lmnt
    }
    ring struct {
        current *lmnt
        len int
    }
)

func NewRing(len int) ring {
    return ring {nil, 0}
}

func (r ring) Empty() bool  {
    return r.Length() == 0
}

func (r ring) Length() int  {
    return r.len
}

func (r ring) GetCurrentValue() int {
```

```
    if r.current == nil {
        return math.MinInt64
    } else {
        return (*r.current).n
    }
}

func (r *ring) RemoveNext () error {
    if r.Empty() {
        return errors.New("Invalid removing")
    }
    if r.Length() > 1 {
       (*(*r).current).next = ((*(*r).current).next).next
    } else {
    // r.Length() == 1
        (*r).current = nil
    }
    (*r).len--
    return nil
}

func (r *ring) PopNext () (int, error) {
    if r.Empty() {
        return 0, errors.New("Invalid removing")
    }
    val:= (*(*(*r).current).next).n
    return val, (*r).RemoveNext()
}

func (r *ring) InsertFront (value int) {
    if r.Empty() {
        (*r).current = new(lmnt)
        (*(*r).current).n = value
        (*(*r).current).next = (*r).current
    } else {
        p:= &lmnt{value, (*(*r).current).next}
        (*(*r).current).next = p
    }
    (*r).len++
}

func (r *ring) OneStep () error {
    if (*r).Empty() {
        return errors.New("Invalid step")
```

```
    }  else  {
        (*r).current = (*(*r).current).next
        return nil
    }
}

func main() {
    var N, M int
    fmt.Print("Enter N: ")
    fmt.Scanf("%d\n", &N)
    fmt.Print("Enter M: ")
    fmt.Scanf("%d\n", &M)

    r:= NewRing(N)
    for i:= N; i > 0; i-- {
        r.InsertFront(i)
    }

    for i:= 0; i < N; i++ {
        for j:= 0; j < (M -1)% r.Length(); j++  { r.OneStep() }
        if tmp, err:= r.PopNext(); err == nil  {fmt.Println(tmp)}
    }
}
```

Задачи и упражнения

А на практику даём всевозможные задания на списки. При этом есть смысл хоть чуть-чуть, по-быстрому так обсудить, а может быть есть смысл вместе с указателем на начало списка хранить его актуальную длину? Чтобы не бегать по всему списку, подсчитывая. С учётом того, что длина нам нужна довольно часто, например, прочитать/вставить/удалить элемент на K-й позиции - обязательно же надо проверять корректность такого действия.

Их полно. Какие-то уже были сформулированы в предыдущем плане. Но простор тут для фантазии неограниченный практически: создать копию списка; создать копию списка, но выкидывая повторяющиеся элементы; просто выбросить из списка повторяющиеся элементы; реализовать какую-нибудь сортировку списка (даже «пузырь» можно, так уж и быть); расщепить список на два – отдельно элементы, стоящие на нечётных местах, отдельно – на чётных. Ну и т.д. Большой набор всяких заданий находится рядом - в файлах list_tasks.md и list_tasks.pdf.