→ LEMBREM-SE DE USAR PAPEL E CANETA COMO RASCUNHO ANTES DE IMPLEMENTAR <<--

# Trabalho 2 - Snake

(a data limite para entrega é informada no Submitty)

Arquivos de apoio disponibilizados:
https://drive.google.com/file/d/1BPUei01k-dqXi6CQdVu_c-RqpGELUsv7/view?usp=sharing


Jogo Snake em um celular antigo da Nokia (fonte: https://www.joe.co.uk/entertainment/iconic-nokia-game-snake-back-play-smartphone-193144)

O Snake (ou Worm) e' um tipo jogo simples e divertido. Desde a década de 70 foram criadas inúmeras variações de tal jogo, sendo que ele foi inclusive um dos primeiros jogos a serem utilizados em telefones celulares.

Para quem não conhece (curiosidade: existe alguém que não conhece?), tal jogo possui um funcionamento muito simples: o jogador controla uma cobra em um ambiente retangular. Em intervalos de tempo (aleatórios) alimentos (um ou mais) aparecem na tela (eles desaparecem quando são comidos ou após um determinado tempo) e o jogador ganha pontos caso consiga comer algum desses alimentos. Para cada ponto obtido o comprimento da cobra aumenta em uma unidade. O jogo termina quando a cobra colide com alguma parede ou com ela mesma.

A versão clássica (que iremos implementar) do jogo pode ser facilmente encontrada na internet (por exemplo, ela está disponivel aqui: https://playsnake.org/ ). Um exemplo de variação divertida do jogo está disponivel aqui: http://slither.io/ (essa versao e' multiplayer).

Embora a versão clássica possa ser implementada com bastante facilidade, iremos adotar uma implementação ineficiente e não muito inteligente (mas que será útil para praticarmos alguns conceitos importantes vistos em INF213).

Leia este roteiro com bastante atenção antes de começar sua implementação. Antes de submeter a versão final do seu trabalho leia este roteiro novamente e confira se sua implementação seguiu toda a especificação.

**Aviso**: apesar desta especificação ter ficado grande (devido aos detalhes), sua implementacao do trabalho provavelmente ficará pequena! Ele é mais simples do que aparenta.

**Aviso 2**: **comece sua implementacao com antecedencia! O deadline dos trabalhos e' firme.**

**Aviso 3**: como um dos objetivos e' praticar o uso de alocação de memórias e ponteiros, seu programa deverá utilizar alocação dinâmica de memória de forma explícita (utilizando os operadores ***new*** e ***delete*** -- i.e., você não pode utilizar classes prontas (como MyVec) para armazenar os dados na versão final do seu trabalho).

## Implementacao

Você deverá seguir a especificação de forma bastante precisa. Os membros de dados de suas classes devem possuir exatamente os nomes descritos aqui (quando mencionados). Além disso, suas classes não poderão ter outros membros de dados (a não ser que você decida criar alguma funcionalidade extra e o professor autorize isso). Você pode (e provavelmente deve) criar funções adicionais (não listadas neste documento) para que sua classe funcione corretamente.

Sua implementação deverá possuir 3 classes (implementadas usando arquivos .h e .cpp): Snake, Game e Screen.

A classe Snake será utilizada para representar a cobra (e terá funções para movê-la e desenhá-la em uma "tela"). A classe Screen representará a tela do jogo (e terá funções para acessar cada "pixel" dessa tela). A clase Game, por sua vez, representará a lógica do jogo (tendo funções para mover a cobra, adicionar comida ao jogo, etc).

A seguir, informações detalhadas (mais detalhes sobre o funcionamento do jogo podem ser inferidos observando exemplos de entrada e saída) sobre cada classe serão providas:

**Classe Game**:
Essa classe representará a lógica do jogo, e armazenará objetos das outras duas classes.

Nessa classe você deverá armazenar os seguintes membros de dados:

- Um objeto do tipo Snake: ou seja, a cobra do jogo.
- Um objeto do tipo Screen: ou seja, a tela do jogo será gerenciada pela classe Game. Tal objeto deverá representar o estado atual da tela do jogo.
- Um array de comida: ou seja, serão armazenadas informações sobre a comida atualmente presente no jogo. **É garantido que nunca teremos mais do que 10 alimentos ativos simultaneamente no jogo.**
- Outros membros de dados podem ser necessários aqui (nesta classe específica você pode adicioná-los se julgar necessário).

Métodos publicos:

- **Construtor** com três parâmetros: altura do jogo a ser criado, largura do jogo e tamanho inicial da cobra (todos valores são medidos em “pixels”).
- Método **getScreen()**: retorna um objeto do tipo Screen representando o estado atual do jogo (tal objeto será utilizado, por exemplo, para exibir o jogo para o usuário).
- Método **step(dr,dc)**: avança com o jogo em uma iteração/unidade de tempo, supondo que (dr,dc) representa o deslocamento (em termos de linha e coluna) escolhido pelo usuário. Por exemplo, se dr=0 e dc=-1 → a cobra deverá se movimentar para trás (em termos de coluna) e se manter na mesma linha. Os únicos deslocamentos válidos são: (-1,0), (1,0), (0,-1), (0,1) (representando as 4 “setas” do teclado direcional). Assuma que apenas deslocamentos válidos serão utilizados. Caso o usuário escolha um movimento que faria com que a cobra inverta sua direção, o método step deverá ignorar esse movimento e manter a direção anterior (exemplo: se a cobra estiver indo para a direita e o usuário tentar movimentá-la para a esquerda → o método step deverá ignorar isso e movimentar a cobra para a direita). Esse método deverá retornar um valor booleano indicando se o movimento foi realizado com sucesso (true) ou não (false) -- um movimento não é realizado com sucesso apenas quando a cobra bate em uma parede ou nela mesma. Ao entrar em uma posição da tela onde ha comida a cobra deverá come-la.
- Método **addFood(r,c,ttl)**: adiciona um alimento na posição (r,c) (linha, coluna) da tela. O terceiro argumento (ttl -- time to live) indica por quanto tempo (em termos de iterações) o alimento deve existir (**após esse tempo ele deve ser removido**). Note que as iteracoes avancam apenas quando o comando step é chamado (cada chamada é uma iteracao). Caso o alimento seja adicionado em uma posição da tela que já possui algo, o método addFood não deve fazer nada (ele deve se comportar como se a última chamada não tivesse sido feita).
- Método **getNumFood()** : retorna um inteiro representando a quantidade de comida (ativa) atualmente no jogo.

**Classe Screen**:

Essa classe representará o estado da tela do jogo (iremos ler os dados de um objeto do tipo screen para desenhar o jogo na tela). É basicamente uma matriz dinâmica representando a tela. Cada objeto da tela é representado por um código (inteiro) inteiro distinto .

Nessa classe você deverá armazenar exatamente os seguintes membros de dados:

- Constantes (públicas) inteiras **FOOD**, **SNAKE**, **EMPTY** e **WALL** que representam, respectivamente, comida, cobra, posição vazia da tela e parede.
- Largura e altura da tela (em termos de número de pixels).
- Uma matriz dinamica "**data**" de inteiros.
- Um vetor "**dataHeight**" de inteiros.

Métodos publicos:

- Construtor com dois parâmetros inteiros representando, respectivamente, a altura e largura da tela do jogo.
- getWidth(), getHeight() : retornam, respectivamente, a largura e a altura da tela do jogo.
- get(r,c) : retorna o estado do "pixel" que está na linha r, coluna c da tela. Consultas a posições fora da tela devem retornar WALL.
- set(r,c,val): armazena na linha r, coluna c da matriz o estado val (que poderá ser: FOOD, SNAKE ou EMPTY). Assuma que a posição (r,c) sempre estará dentro da tela.

A classe Screen poderia ser armazenada simplesmente como uma matriz retangular. Porém, para deixar este trabalho mais desafiador (e envolver mais estrutura de dados) **você deverá utilizar a seguinte estratégia**:

- data[c] deverá apontar para um array dinâmico representando a "c-ésima" coluna do jogo.
- dataHeight[c] deverá conter 1 + a linha do pixel não vazio mais alto da tela (ou 0, caso não tenha pixels na coluna).
- O tamanho do array data[c] deverá ser dataHeight[c].
- Ou seja, a matriz representando a tela deverá ser armazenada de forma compacta e operações que chamam a função "set" na coluna "c" devem redimensionar a matriz dinamicamente para que o elemento mais alto que não for EMPTY da coluna "c" deva sempre estar armazenado na última posição de data[c].

**Classe Snake**:
A classe Snake armazenará a cobra do jogo utilizando uma lista encadeada (você deverá decidir entre duplamente ou simplesmente encadeada).

Nessa classe você deverá armazenar exatamente os seguintes membros de dados:

- Ponteiros para o primeiro e último nodo da lista.

Métodos publicos:

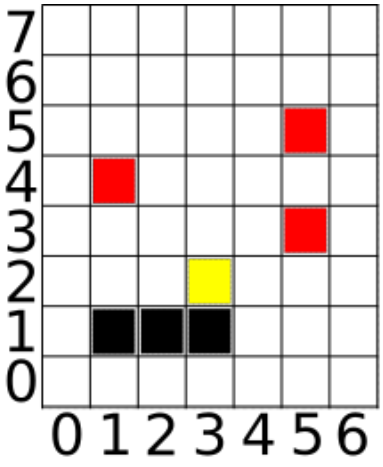
- Método draw(s,state): desenha a cobra no objeto (da classe Screen) s, marcando as posições da cobra com o identificador "state" (um inteiro). Por exemplo, se state for igual a 10 → o método draw irá utilizar o método "set" de s para setar todas posições relativas à cobra com o valor 10.

- Método move(dr, dc, eating): move a cabeça da cobra na direção (dr,dc) (é garantido que esse par de valores terá apenas uma das 4 possíveis combinações mencionadas na classe “Game”). Por exemplo, se (dr,dc) = (1,0) a cabeca da cobra moverá um pixel para a próxima linha e se manterá na mesma coluna. “eating” é um booleano que indica se a cobra está comendo ou não durante o movimento (ou seja, essa variável será true se a cabeça da cobra for se mover para um pixel que atualmente contém comida). Quando a cobra come um alimento seu último pixel não se move durante esse movimento. **O método move deverá ter complexidade O(1)**.
- getLength(): retorna o tamanho da cobra (em “pixels”).
- Construtor com um parâmetro inteiro: esse parâmetro é o tamanho inicial do jogador. Ao construir uma cobra de tamanho N suas coordenadas **sempre** serão: (0,0), (0,1), (0,2), … (0,N-1) (a cabeça estará na posição (0,N-1) ). É garantido que o parâmetro sempre será maior ou igual a 1 e que nunca tentaremos construir uma cobra que inicialmente não cabe na largura da tela.

A Figura a seguir ilustra o estado de um jogo com 8 linhas e 7 colunas. A grade superior ilustra a tela do jogo (pixels em vermelho são comida e pixels em preto são a cobra -- note que a cabeça foi pintada de amarelo apenas para destacar o sentido do jogador) e a parte inferior exibe o estado da memória.

Dentro do objeto Game temos o array de comida (por exemplo o alimento na posição (4,1) tem “tempo de vida” 3 -- ou seja, deverá desaparecer na quarta iteração). Além disso, temos um objeto Snake com a lista encadeada representando uma cobra com 4 pixels. Por fim, temos também um objeto Screen (note que “data” utiliza a representação compacta mencionada acima para tentar “economizar memória”).

É recomendado que o aluno tente desenhar diagramas similares ao abaixo para entender o funcionamento do jogo. Por exemplo, tente desenhar os diagramas obtidos em 3 iterações do jogo.




# Game

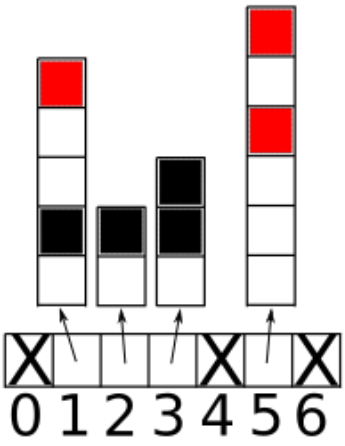
## Screen

width=7
height=8

dataHeight

| 0 | 5 | 2 | 3 | 0 | 6 | 0 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

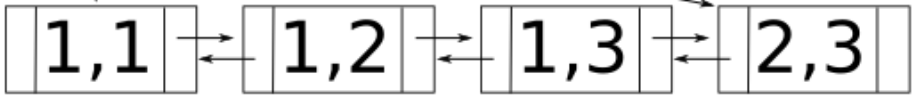
data




## Snake

last

first




## Comida

| 4,1,3 | 5,5,3 | 5,3,2 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|

## Jogo utilizando sua classe

O arquivo **jogoSnake.cpp** contém uma implementação de interface com o usuário utilizando a biblioteca ncurses. Ou seja, se você compilar tal arquivo juntamente com suas classes terá um jogo.

Para compilar tal arquivo, é necessário ter a biblioteca ncurses (que pode ser facilmente instalada pelo gerenciador de pacotes de uma distribuição Linux) e "linka-la" ao seus arquivos objeto (exemplo: "g++ jogoSnake.cpp Snake.cpp Game.cpp Screen.cpp -lncurses -o jogoSnake.exe").

Note que essa implementação utiliza uma interface de texto. Uma interface gráfica poderia facilmente ser criada utilizando, por exemplo, a biblioteca Allegro.

## Jogo snake a partir de comandos

O arquivo jogoSnakeAvaliaComandos.cpp será utilizado pelo Submitty para fazer alguns testes em suas classes. Tal programa utiliza comandos de texto para comandar o jogo. Você pode utilizá-lo para realizar testes em suas classes.

Por exemplo, a entrada abaixo cria um jogo com 5 linhas , 6 colunas e tamanho inicial do jogador com 3 pixels. A seguir, alguns alimentos são adicionados à tela e alguns movimentos são realizados (exemplos de entrada e saída esperadas estão disponíveis juntamente com os arquivos fonte de exemplo). Note que não da tempo do jogador comer o primeiro alimento.

Entrada (exemplo0.txt):

```
5 6 3
food 0 4 1
food 2 2 10
step 0 1
step 0 1
food 4 4 10
step 1 0
```

Nesse outro exemplo, por outro lado, o jogador chega na comida 0,4 no tempo 2 (a comida só é removida após o tempo 2) e, portanto, ele consegue se alimentar.

Entrada (exemplo1.txt):

```
5 6 3
food 0 4 2
food 2 2 10
step 0 1
step 0 1
food 4 4 10
step 1 0
```

Note que, após comer uma comida a cobra “passa por cima dela” e, portanto, após sair da posição onde havia comida o alimento não deverá continuar lá mais.

Entrada (exemplo2.txt):
5 6 3
food 0 4 20
food 2 2 10
step 0 1
step 0 1
food 4 4 10
step 1 0
step 1 0
step 1 0
step 0 -1
step 0 -1

## Problemas na estratégia utilizada

Há **MUITAS** formas melhores de se implementar o jogo Snake, porém, não seguiremos essas estratégias mais adequadas por alguns motivos.

Primeiro, alguns conceitos que poderiam ser úteis ainda não foram estudados na disciplina (ou foram estudados muito recentemente). Por exemplo, idealmente a classe Snake não deveria ter a função draw: em vez disso, o jogo deveria utilizar um iterador para iterar por todos as coordenadas que formam a cobra.

Segundo, a implementação proposta tem como objetivo apenas a prática de conceitos importantes de estruturas de dados. Se algumas soluções (melhores) fossem seguidas, tais conceitos não seriam muito estudados. Por exemplo, não faz muito sentido ter a classe Screen (os elementos do jogo poderiam ser desenhados diretamente na tela) e, além disso, a “economia” de memória que obtemos com essa classe não é relevante nesse jogo.

Além disso, o uso de estruturas de dados prontas (disponíveis na STL do C++, por exemplo) facilitaria enormemente toda a implementação (normalmente reutilizamos estruturas de dados prontas em vez de implementá-las “manualmente”). Porém, se simplesmente reutilizássemos tais estruturas vários conceitos importantes não seriam praticados neste trabalho.

## Arquivo README

Seu trabalho deverá incluir um arquivo README.

Tal arquivo contera:

- Seu nome/matricula

- Informacoes sobre todas fontes de consulta utilizadas no trabalho

**Submissao**

Submeta seu trabalho utilizando o sistema Submitty até a data limite. Seu programa será avaliado de forma automática (os resultados precisam estar corretos, o programa não pode ter erros de memória, etc), passará por testes automáticos "escondidos" e a qualidade do seu código será avaliada de forma manual.

Você deverá enviar os arquivos: README, seus arquivos .cpp e .h (com os nomes definidos neste documento).

**Duvidas**

Dúvidas sobre este trabalho deverão ser postadas no sistema Piazza. Se esforce para implementá-lo e não hesite em postar suas dúvidas!

**Avaliacao manual**

Principais itens que serão avaliados (além dos avaliados nos testes automáticos):
- Comentarios
- Indentacao
- Nomes adequados para variáveis
- Separação do código em funções lógicas
- Uso correto de const/referencia
- Uso de variáveis globais apenas quando absolutamente necessário e justificável (uso de variáveis globais, em geral, e' uma má prática de programação).
- etc

**Regras sobre plágio e trabalho em equipe**

- Este trabalho deverá ser feito de forma individual.
- Porém, os alunos podem ler o roteiro e discutir ideias/algoritmos em alto nível (nível mais alto do que pseudocódigo) de forma colaborativa.
- As implementações (nem mesmo pequenos trechos de código) não deverão ser compartilhadas entre alunos. Um estudante não deve olhar para o código de outra pessoa.
- Um estudante não deve utilizar o computador de outro colega para fazer/submeter o trabalho (devido ao risco de um ter acesso ao código do outro). Cuidado para não deixar seu código fonte em algum computador público (a responsabilidade pelo seu código e' sua).
- Crie um arquivo README (submeta-o com o trabalho) e inclua todas as suas fontes de consulta (incluindo pessoas que lhe ajudaram em algo do trabalho).
- Não poste seu código (nem parte dele) no Piazza (ou outros sites) de forma pública (cada aluno e' responsável por evitar que outros plagiem seu código).
- Se você estiver lendo isto (deveria estar…) escreva "Eu li as regras" na primeira linha do seu README.

- Trechos de código não devem ser copiados de livros/internet. Se consultar algum livro ou material na internet essa fonte deverá ser citada no README.
- Se for detectado plágio (mesmo em pequenos trechos de código) a nota de TODOS estudantes envolvidos será 0.
- Além disso, os estudantes poderão ser denunciados aos órgãos responsáveis por plágio da UFV. Lembrem-se que um conceito F (fraude académica) no histórico escolar pode afetar severamente seu currículo.
- O plágio pode ser detectado após a liberação das notas do trabalho (ou seja, pode ser que o professor reavalie os trabalhos procurando por plágio no final do semestre). Assim, sua nota poderá ser alterada para 0 caso algum problema seja encontrado posteriormente (essa alteração na nota vale apenas para casos de plágio).