Informações gerais

Neste laboratório, você será introduzido ao conceito e prática de Binary Search Tree e Pilha, utilizando-as para encontrar o melhor caminho através de um jogo que consiste em derrotar monstros em uma floresta e coletar suas recompensas.

Você está prestes a enfrentar uma floresta assombrada cheia de terríveis monstros. A primeira linha indica quantos monstros estão presentes na floresta. As próximas linhas representam os poderes dos monstros na ordem em que você os encontrará. Cada confronto com um monstro reduzirá sua vida pela quantidade equivalente ao poder do monstro. À medida que derrota mais monstros, você ganha mais recompensas. Utilize uma Binary Search Tree para organizar os poderes dos monstros e determine quantos caminhos você pode seguir para sair vivo derrotando o máximo de monstros possível.

Observações importantes:

1. Neste laboratório será permitido o uso apenas das bibliotecas `stdio.h`, `stdlib.h` e `string.h`.
2. Para compilar sua solução, utilize o `Makefile`, chamando no terminal o comando `make`.

TAD para manipulação da árvore e da pilha

O laboratório proposto é uma jornada através de uma floresta assombrada, onde o jogador encontra vários monstros. A saúde do jogador diminui com cada encontro, dependendo do poder do monstro. Usando a BST e a pilha, o objetivo é encontrar o caminho que permite ao jogador derrotar o máximo de monstros e ainda assim sobreviver.

O programa lê os poderes dos monstros e o poder do jogador, constrói a BST e, em seguida, explora todos os caminhos possíveis. Para cada caminho viável, os monstros derrotados são armazenados na pilha, que é então impressa, mostrando o sucesso do jogador. A TAD para manipular a Árvore de Busca Binária (Binary Search Tree) e Pilha terá os seguintes operadores:

- **cria percursos**: Esta função inicializa a árvore, garantindo que ela esteja vazia antes de começar a inserção dos monstros.
- **insere monstro**: Aqui, os monstros são inseridos na árvore. A inserção segue a lógica da BST, onde cada novo monstro (nó) é colocado à esquerda se seu poder for menor que o do nó atual e à direita se for maior.
- **busca caminhos**: Esta função é crucial, pois procura na árvore todos os caminhos viáveis (onde o jogador permanece vivo) com base no poder total do jogador. Ela deve navegar pela árvore, calculando a soma dos poderes dos monstros em cada caminho e comparando com o poder do jogador.
- **destrói percursos**: Para garantir que não haja vazamento de memória, esta função é usada para liberar toda a memória alocada para a árvore após o seu uso.

Para manipular a pilha, implementaremos os seguintes operadores:

- **cria pilha**: Inicia uma nova pilha de monstros.
- **imprime pilha**: Após encontrar um caminho viável, a pilha de monstros (representando o caminho) é impressa usando esta função.
- **insere monstro**: Quando um caminho está sendo explorado e um novo monstro é enfrentado, ele é adicionado à pilha.
- **remove monstro**: Se um caminho se provar inviável (por exemplo, o jogador não tem poder suficiente para um monstro), o monstro é removido da pilha, e o jogo retorna para o ponto de decisão anterior.

- **destrói pilha**: Como na árvore, essa função é crucial para evitar vazamentos de memória, destruindo a pilha após seu uso.

Questão 1

Nesta questão, implementaremos o arquivo `cliente.c` para ler da entrada padrão uma sequência de comandos. Os comandos são os seguintes:

- A primeira linha contém o número **N** de monstros.
- As **N** linhas seguintes possuem o poder de cada monstro, que deve ser inserido na Binary Search Tree na ordem que aparecem.
- A última linha contém o nível de poder do seu personagem.

Seu programa deve construir a Árvore de Busca Binária (Binary Search Tree) com precisão, assegurando que os poderes dos monstros sejam inseridos corretamente conforme a sequência fornecida.

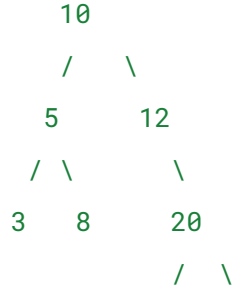
Ademais, é necessário identificar todos os caminhos possíveis que resultem no maior número de recompensas, garantindo que seu personagem sobreviva (isto é, mantendo a vida em um saldo positivo). Paralelamente, é imprescindível organizar uma pilha que contenha os monstros encontrados ao longo dos caminhos bem-sucedidos.

Exemplo de entrada e saída

Se for fornecida a seguinte sequência de comandos:

```
8
10
5
12
3
8
20
15
30
25
```

A seguinte árvore será construída:



```
        15    30
```

Se o poder do seu personagem for **25**, os seguintes caminhos serão possíveis:

```
2
3, 5, 10
8, 5, 10
```

Ambos os percursos contêm três monstros derrotados. A sequência para impressão deve seguir a ordem estabelecida na pilha, listando os caminhos do menor para o maior total acumulado.

Agora, se a última linha indicasse um poder de **60**, conforme o exemplo abaixo:

```
8
10
5
12
3
8
20
15
30
60
```

A saída seria:

```
1
15, 20, 12, 10
```

A estrutura da árvore permanece inalterada, porém, haverá apenas um percurso viável **(10, 12, 20, 15)**, no qual é possível derrotar quatro monstros, um número superior ao do exemplo anterior.

Após codificar o cliente e também as operações necessárias na implementação, compile tudo utilizando o `Makefile` e teste executando o seguinte comando:

```
./cliente.bin < teste_Q1.in
```

Onde `teste_Q1.in` é um arquivo contendo uma sequência de comandos como a exemplificada acima.