Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Herança

Problema a resolver

O tipo sanguíneo de uma pessoa é determinado por dois alelos (ou seja, diferentes formas de um gene). Os três alelos possíveis são A, B e O, dos quais cada pessoa possui dois

(possivelmente iguais, possivelmente diferentes). Cada um dos pais de uma criança transmite aleatoriamente um de seus dois alelos do tipo sanguíneo para o filho. As possíveis combinações de tipos sanguíneos são, portanto,: OO, OA, OB, AO, AA, AB, BO, BA e BB.

Por exemplo, se um dos pais tem sangue tipo AO e o outro tem sangue tipo BB, os possíveis tipos sanguíneos da criança seriam AB e OB, dependendo do alelo recebido de cada um. Da mesma forma, se um dos pais tem sangue tipo AO e o outro OB, os possíveis tipos sanguíneos da criança seriam AO, OB, AB e OO.

Em um arquivo chamado localizado `inheritance.c` em uma pasta chamada `inheritance`, simule a herança dos tipos sanguíneos para cada membro de uma família.

Demonstração

```
    Parent (Generation 1): blood type OO
        Grandparent (Generation 2): blood type BO
        Grandparent (Generation 2): blood type OA
$ ./inheritance
Child (Generation 0): blood type AO
    Parent (Generation 1): blood type AA
        Grandparent (Generation 2): blood type AA
        Grandparent (Generation 2): blood type AA
    Parent (Generation 1): blood type AO
        Grandparent (Generation 2): blood type AO
        Grandparent (Generation 2): blood type OO
$
```

Recorded with **asciinema**

Código de Distribuição

Para este problema, você deverá estender a funcionalidade do código fornecido pela equipe do CS50.

▼ **Baixe o código de distribuição.**

Faça login em cs50.dev (https://cs50.dev/) , clique na janela do terminal e execute `cd` o comando. Você verá que o prompt do terminal será semelhante ao seguinte:

```
$
```

Em seguida, execute

```
wget https://cdn.cs50.net/2024/fall/psets/5/inheritance.zip
```

para baixar um arquivo ZIP chamado `inheritance.zip` para o seu espaço de código.

Em seguida, execute

```
unzip inheritance.zip
```

para criar uma pasta chamada `inheritance`. Você não precisa mais do arquivo ZIP, então pode executar

```
rm inheritance.zip
```

e responda com "y" seguido de Enter quando solicitado para remover o arquivo ZIP que você baixou.

Agora digite

```
cd inheritance
```

Em seguida, pressione Enter para entrar (ou seja, abrir) esse diretório. Seu prompt agora deve ser semelhante ao abaixo.

```
inheritance/ $
```

Execute o comando `ls` sozinho e você deverá ver um arquivo chamado `inheritance.c`.

Se você encontrar algum problema, siga esses mesmos passos novamente e veja se consegue determinar onde errou!

Detalhes da implementação

Complete a implementação de `inheritance.c` forma que ela crie uma família com um tamanho de geração especificado e atribua alelos de tipo sanguíneo a cada membro da família. A geração mais velha terá alelos atribuídos aleatoriamente.

- A `create_family` função recebe um inteiro (`generations`) como entrada e deve alocar (como via `malloc`) um `person` para cada membro da família desse número de gerações, retornando um ponteiro para na `person` geração mais jovem.
  - Por exemplo, `create_family(3)` deve retornar um ponteiro para uma pessoa com dois pais, onde cada pai também tem dois pais.
  - Cada geração `person` deve ter `alleles` alelos atribuídos. A geração mais velha deve ter alelos escolhidos aleatoriamente (por exemplo, chamando a `random_allele` função), e as gerações mais jovens devem herdar um alelo (escolhido aleatoriamente) de cada progenitor.

- Cada um `person` deve ter `parents` um valor atribuído a eles. A geração mais antiga deve ter ambos os `parents` valores definidos como `NULL`, e as gerações mais jovens devem ter `parents` um array de dois ponteiros, cada um apontando para um pai diferente.

Dicas

Clique nos botões abaixo para ler algumas dicas!

▼ **Entenda o código em`inheritance.c`**

Dê uma olhada no código de distribuição em `inheritance.c`.

Observe a definição de um tipo chamado `person`. Cada pessoa possui um array de dois `parents`, cada um dos quais é um ponteiro para outra `person` struct. Cada pessoa também possui um array de dois `alleles`, cada um dos quais é um `char` (seja `'A'`, `'B'`, ou `'O'`).

```
// Each person has two parents and two alleles
typedef struct person
{
    struct person *parents[2];
    char alleles[2];
}
person;
```

Agora, observe a `main` função. A função começa "inicializando" (ou seja, fornecendo uma entrada inicial para) um gerador de números aleatórios, que usaremos posteriormente para gerar alelos aleatórios.

```
// Seed random number generator
srandom(time(0));
```

A `main` função então chama a `create_family` função para simular a criação de `person` estruturas para uma família de 3 gerações (ou seja, uma pessoa, seus pais e seus avós).

```
// Create a new family with three generations
person *p = create_family(GENERATIONS);
```

Em seguida, solicitamos `print_family` a impressão dos dados de cada um desses membros da família e seus respectivos tipos sanguíneos.

```
// Print family tree of blood types
print_family(p, 0);
```

Finalmente, a função chama `free_family` qualquer `free` memória que tenha sido alocada anteriormente com `malloc`.

```
// Free memory
free_family(p);
```

As funções `create_family` `and` `free_family` ficam a seu critério!

▼ **Complete a `create_family`função**

A `create_family` função deve retornar um ponteiro para um `person` objeto que herdou o tipo sanguíneo a partir do número de `generations` objetos fornecidos como entrada.

- Note, em primeiro lugar, que este problema oferece uma boa oportunidade para recursão.
  - Para determinar o tipo sanguíneo da pessoa em questão, é necessário primeiro determinar o tipo sanguíneo de seus pais.
  - Para determinar o tipo sanguíneo dos pais, primeiro você precisa determinar o tipo sanguíneo dos pais *deles* . E assim por diante, até chegar à última geração que deseja simular.

Para resolver esse problema, você encontrará vários itens TODO no código de distribuição.

Primeiro, você deve alocar memória para uma nova pessoa. Lembre-se de que você pode usar `mk` `malloc` para alocar memória e `sizeof(person)` `mk` para obter o número de bytes a serem alocados.

```
// Allocate memory for new person
person *new_person = malloc(sizeof(person));
```

Em seguida, você deve verificar se ainda há gerações a serem criadas: ou seja, se `generations > 1` ...

Se `generations > 1` , então ainda há mais gerações que precisam ser alocadas. Já criamos dois novos pais, `parent0` e `parent1` , chamando recursivamente `create_family` . Sua `create_family` função deve então definir os ponteiros parentais da nova pessoa que você criou. Finalmente, atribua ambos `alleles` para a nova pessoa, escolhendo aleatoriamente um alelo de cada pai.

- Lembre-se, para acessar uma variável por meio de um ponteiro, você pode usar a notação de seta. Por exemplo, se `x` `p` for um ponteiro para uma pessoa, então um ponteiro para o primeiro pai dessa pessoa pode ser acessado por ` `p->parents[0]` x`.
- Você pode achar a `random()` função útil para atribuir alelos aleatoriamente. Esta função retorna um número inteiro entre 0 `0` e 1 `RAND_MAX` , ou `32767` 0. Em particular, para gerar um número pseudoaleatório que seja 0 `0` ou 1 `1` , você pode usar a expressão 1 = 0 `random() % 2` .

```
// Create two new parents for current person by recursively calling create_family
person *parent0 = create_family(generations - 1);
person *parent1 = create_family(generations - 1);
```

```
// Set parent pointers for current person
new_person->parents[0] = parent0;
new_person->parents[1] = parent1;

// Randomly assign current person's alleles based on the alleles of their parents
new_person->alleles[0] = parent0->alleles[random() % 2];
new_person->alleles[1] = parent1->alleles[random() % 2];
```

Suponhamos que não haja mais gerações para simular. Ou seja, `generations == 1`. Nesse caso, não haverá dados dos pais para essa pessoa. Ambos os pais da sua nova pessoa devem ser definidos como `NULL`, e cada um `allele` deve ser gerado aleatoriamente.

```
// Set parent pointers to NULL
new_person->parents[0] = NULL;
new_person->parents[1] = NULL;

// Randomly assign alleles
new_person->alleles[0] = random_allele();
new_person->alleles[1] = random_allele();
```

Por fim, sua função deve retornar um ponteiro para a `person` variável que foi alocada.

```
// Return newly created person
return new_person;
```

▼ **Complete a `free_family` função**

A `free_family` função deve aceitar como entrada um ponteiro para uma pessoa `person`, liberar memória para essa pessoa e, em seguida, liberar recursivamente a memória para todos os seus ancestrais.

- Como esta é uma função recursiva, você deve primeiro tratar o caso base. Se a entrada da função for `NULL`, então não há nada para liberar, portanto sua função pode retornar imediatamente.
- Caso contrário, você deve consultar recursivamente `free` ambos os pais da pessoa antes de `free` consultar a criança.

A dica abaixo é bastante sugestiva, mas aqui está como fazer exatamente isso!

```
// Free `p` and all ancestors of `p`.
void free_family(person *p)
{
    // Handle base case
    if (p == NULL)
    {
        return;
    }

    // Free parents recursively
    free_family(p->parents[0]);
    free_family(p->parents[1]);
```

```
    // Free child
    free(p);
}
```

Passo a passo

▶ **Não sabe como resolver?**

Como testar

Ao executar `./inheritance` o programa, ele deverá seguir as regras descritas no plano de fundo. A criança deverá ter dois alelos, um de cada progenitor. Os progenitores deverão ter dois alelos cada, um de cada um de seus pais.

Por exemplo, no exemplo abaixo, a criança da Geração 0 recebeu um alelo O de ambos os pais da Geração 1. O primeiro pai recebeu um alelo A do primeiro avô e um alelo O do segundo avô. Da mesma forma, o segundo pai recebeu um alelo O e um alelo B de seus avós.

```
$ ./inheritance
Child (Generation 0): blood type OO
    Parent (Generation 1): blood type AO
        Grandparent (Generation 2): blood type OA
        Grandparent (Generation 2): blood type BO
    Parent (Generation 1): blood type OB
        Grandparent (Generation 2): blood type AO
        Grandparent (Generation 2): blood type BO
```

Correção

```
check50 cs50/problems/2025/x/inheritance
```

Estilo

```
style50 inheritance.c
```

Como enviar

```
submit50 cs50/problems/2025/x/inheritance
```