Estruturas de Dados I

Sacos

Igor Machado Coelho

29/03/2021

Section 1

Sacos e Listas

Pré-Requisitos

São requisitos para essa aula o conhecimento de:

- Introdução/Fundamentos de Programação (em alguma linguagem de programação)
- Interesse em aprender C/C++
- Noções de tipos de dados

Material Complementar

Materiais complementares:

- Livro: R. Sedgewick and K. Wayne, Algorithms, 4th Edition, Addison-Wesley, 2011
- Códigos em Java: https://algs4.cs.princeton.edu/code/

Section 2

Tipo Abstrato: Saco

Saco

O *Saco* ou *Bolsa* (do inglês *Bag*) é um Tipo Abstrato de Dado (TAD) que serve para agregar elementos de um tipo pré-definido (estrutura homogênea).

Em um *saco de produtos*, por exemplo:

- Se consegue "inserir" (do inglês *add*) novos produtos
- Podemos "percorrer" ou "iterar" (do inglês *iterate*) sobre os produtos
- Podemos "buscar" ou "encontrar" (do inglês *find*) um produto específico
- Podemos "remover" (do inglês *remove*) um dado produto

O *Saco* é também conhecido como *multiset* e alguns autores não consideram operações de remoção. [1]

[1] De acordo com NIST *"An unordered collection of values that may have duplicates"*.

Saco na computação

Sacos são estruturas fundamentais na própria computação, como o conceito de `multiset`. Diferente de um `set`, não se exige que elementos sejam únicos.

Em Python, a classe `Counter` representa um tipo de saco. Em `Java`, a interface Bag representa um saco. Em C++, pode-se considerar containers como `std::list` e `std::vector`.

Operações de um Saco (TAD)

Um Saco é um tipo abstrato de dado (TAD) que consiste de 4 operações básicas:

- adicionar (*add*)
- iterar (*iterate*)
- buscar (*find*)
- remover (*remove*)

Note que o Saco não tem conceito de *indexação* nem uma ordem específica de percurso / ordenação dos elementos. Elementos podem ser repetidos.

Implementações

Veja operações: `adiciona('c'); adiciona('a'); adiciona('b'); adiciona('a');`

Podemos responder as perguntas: *'a' existe? 'd' existe?*

Quais estruturas de dados podem ser usadas para implementar as operações de um Saco?

Apresentaremos inicialmente a implementação do TAD Saco através de Listas Encadeadas, embora também possa ser feito através de Arrays.

Definição do *Conceito* Saco em C++

O *conceito* de saco somente requer suas operações básicas. Como consideramos um *saco genérica* (saco de inteiro, char, etc), definimos um *conceito genérico* chamado `SacoTAD`:

```
template <typename Agregado>
concept bool
SacoTAD = requires(Agregado a, typename Agregado::Tipo t,
                   typename Agregado::ItTipo it) {
    // requer operação 'adiciona' sobre tipo 't'
    {a.adiciona(t)};
    // requer operação 'itera' (retorna 'it')
    {a.itera()};
    // requer operação 'busca' sobre tipo 't' (retorna 'it')
    {a.busca(t)};
    // requer operação 'remove'
    {a.remove(it)};
};
```

Utilização do SacoTAD

Antes de completar as funções pendentes, utilizaremos um SacoTAD:

```
int main () {
    SacoTAD s = ...; // alguma implementação
    s.cria();
    s.adiciona('c');
    s.adiciona('a');
    s.adiciona('b');
    s.adiciona('a');
    printf("%d\n", s.busca('a').terminou()); // 0
    printf("%d\n", s.busca('d').terminou()); // 1
    for (auto it = s.itera(); !it.terminou(); it.proximo())
        printf("%c\n", it.atual()); // a b a c (em qual ordem?)
    s.remove(s.busca('b'));
    s.libera();
    ...
```

Section 3

Lista Encadeada

Lista Encadeada

As listas encadeadas possibilitam a inclusão de elementos em quantidade arbitrária (limitada à memória do computador), através de nós de encadeamento.

Veja operações: `adiciona('c'); adiciona('a'); adiciona('b'); adiciona('a');`

$$[inicio \rightarrow][a| \rightarrow] \ [b| \rightarrow] \ [a| \rightarrow] \ [c| \rightarrow 0]$$

Implementação (I)

Consideraremos uma lista encadeada de caracteres. Para tal, definimos um agregado que contenha elementos, e outro que represente um iterador.

```
class NoEnc1
{
public:
    char dado;
    NoEnc1* proximo;
};

class IteradorNoEnc1
{
public:
    NoEnc1* no;        // no atual
    char atual();      // inglês 'current'
    bool terminou();   // inglês 'isDone'
    void proximo();    // inglês 'next'
};
```

Implementação (II)

```
class ListaEnc1
{
public:
    typedef char Tipo;
    typedef IteradorNoEnc1 ItTipo;

    NoEnc1* inicio; // inicio da lista
    int N;          // num. de elementos na lista
    void cria();    // inicializa agregado
    void libera();  // finaliza agregado
    void adiciona(Tipo dado);
    ItTipo itera();
    ItTipo busca(Tipo dado);
    void remove(ItTipo it);
};
// verifica se agregado ListaEnc1 satisfaz conceito SacoTAD
static_assert(SacoTAD<ListaEnc1>);
```

Utilização da ListaEnc1

Antes de completar as funções pendentes, utilizaremos a `ListaEnc1`:

```
int main () {
   ListaEnc1 l;
   l.cria();
   l.adiciona('c');
   l.adiciona('a');
   l.adiciona('b');
   l.adiciona('a');
   printf("%d\n", l.busca('a').terminou());
   printf("%d\n", l.busca('d').terminou());
   for(auto it = l.itera(); !it.terminou(); it.proximo()) {
      printf("%c\n", it.atual());
   }
   l.libera();

   ...
```

Implementação: Cria e Libera

A operação `cria` inicializa a estrutura para uso, e a função `libera` desaloca os recursos dinâmicos.

```
class ListaEnc1 {
...
void cria() {
    this->N = 0;
}

void libera() {
    while (this->inicio != 0) {
        auto* prox = this->inicio->proximo;
        delete this->inicio;
        this->inicio = prox;
    }
    this->N = 0;
}
```

Implementação: Adiciona

A operação `adiciona` em uma lista encadeada adiciona um novo elemento na "cabeça" da lista (no início).

```
class ListaEnc1 {
...
void adiciona(char dado) {
    auto* no = new NoEnc1{.dado = dado, .proximo = inicio};
    this->inicio = no;
    this->N++; // N = N + 1
}
...
}
```

Conceito: IteradorTAD

Podemos definir um tipo abstrato para o iterador, denominado IteradorTAD.

```
template <typename Agregado>
concept bool
#endif
        IteradorTAD = requires(Agregado a)
{
    // requer operação 'terminou' (retorna booleano)
    {a.terminou()};
    // requer operação 'atual' (retorna elemento)
    {a.atual()};
    // requer operação 'proximo'
    {a.proximo()};
};
```

Implementação: Iterador

As operações do iterador utilizam um marcador/sentinela onde o ponteiro atual vale zero quando não existe mais um próximo.

```
class IteradorNoEnc1 {
    NoEnc1* no;
    char atual() {
        return this->no->dado;
    }
    bool terminou() {
        return this->no == 0;
    }
    void proximo() {
        this->no = this->no->proximo;
    }
}
// verifica se agregado IteradorNoEnc1 satisfaz conceito Itera
static_assert(IteradorTAD<IteradorNoEnc1>);
```

Implementação: Itera

A operação itera em uma lista encadeada retorna o iterador da lista, posicionado no começo.

```
class ListaEnc1 {
...
IteradorNoEnc1 itera() {
    IteradorNoEnc1 it{.no = this->inicio};
    return it;
}
...
}
```

Implementação: Busca

A operação busca em uma lista encadeada utiliza o iterador da lista, posicionado no começo, para encontrar o elemento, retornando o iterador atualizado na posição correspondente.

```
class ListaEnc1 {
...
IteradorNoEnc1 busca(char dado) {
    auto it = this->itera();
    while (!it.terminou())
    {
        if (it.atual() == dado)
            return it;
        it.proximo();
    }
    return it;
}
...
```

Implementação: Busca Recursiva

A busca pode ser feita de forma recursiva também:

```
IteradorNoEnc1 buscarec(IteradorNoEnc1 it, char dado)
{
   if(it.terminou() || it.atual() == dado)
      return it;
   else
   {
      it.proximo();
      return buscarec(it, dado);
   }
}
```

Implementação: Remoção

A remoção de um elemento ocorre a partir de um iterador.

```
class ListaEnc1 {
...
void remove(IteradorNoEnc1 it)
{
    auto *prox = it.no->proximo;
    (*it.no) = (*it.no->proximo); // sobrescrita
    delete prox;
    this->N--; // N = N - 1
}
...
}
```

Section 4

Extensões de Listas Encadeadas

Extensões de Listas Encadeadas

A implementação de listas encadeadas tipicamente permite acesso bidirecional, com um ponteiro `anterior` além de um `proximo`. Este tipo de lista é chamado de *Lista Duplamente Encadeada*.

```
class NoDuploEnc1
{
public:
    char dado;
    NoDuploEnc1* anterior;
    NoDuploEnc1* proximo;
};
```

Desafio: Implemente as operações da lista!

Dica: para simplificar operações nos extremos da lista, pode-se utilizar um nó sentinela de Cabeça e Cauda.

Iteradores de *Range* em C++

É possível transformar a implementação da `ListaEnc1` para uso em *for* do tipo *range*, em C++. Exemplo:

```
int main() {
    SacoTAD s = ListaEnc1();
    s.cria();
    s.adiciona('c');
    s.adiciona('a');
    s.adiciona('b');
    s.adiciona('a');
    printf("%d\n", s.busca('a').terminou()); // 0
    printf("%d\n", s.busca('d').terminou()); // 1
    printf("%d\n", buscarec(s.itera(), 'd').terminou()); // 1
    for (auto x : s)
        printf("%c\n", x); // a b a c
    ...
```

Implementação: Iterador de *Range*

Por padrão, a linguagem C++ exige os métodos `begin()` e `end()` para inicio e fim do processo de iteração, no agregado `ListaEnc1`. Por outro lado, os métodos `terminou()`, `proximo()` e `atual()` são substituídos pelos operadores ==, ++ e *, no agregado `IteradorNoEnc1`.

```
class ListaEnc1
{
...
    // iterador de início
    IteradorNoEnc1 begin() {
        return itera();
    }

    // sentinela de final da iteração
    IteradorNoEnc1 end() {
        return IteradorNoEnc1{.no = 0};
    }
}
```

Implementação de Operadores no Iterador

Tópico Avançado: os métodos `terminou()`, `proximo()` e `atual()` são substituídos pelos operadores ==, ++ e *, no agregado `IteradorNoEnc1`.

```
class IteradorNoEnc1
{
public:
   NoEnc1* no;     // elemento atual
   ...             // demais métodos omitidos

   char operator*() { return atual(); }

   bool operator!=(const IteradorNoEnc1& other) {
      return no != other.no;
   }

   void operator++() { proximo(); }
};
```

Fim implementações (listas)

Fim parte de implementações de listas.

Section 5

Vetores como Sacos

Vetores como Sacos

Os vetores podem ser utilizados como sacos, basta incluir os elementos sequencialmente à medida que forem adicionados.

Veja operações: `adiciona('c'); adiciona('a'); adiciona('b'); adiciona('a');`

$$|c|a|b|a|$$

$$0\ 1\ 2\ 3$$

Implementação

Consideraremos um vetor de caracteres pré-alocado com capacidade `MAX_N`.

```
constexpr int MAX_N = 10000;
class SacoVetor1 {
public:
    typedef char Tipo;
    typedef IteradorVetor1 ItTipo;

    Tipo elementos[MAX_N]; // elementos
    int N;                 // num. de elementos na lista
    void cria();           // inicializa agregado
    void libera();         // finaliza agregado
    void adiciona(Tipo dado);
    ItTipo itera();
    ItTipo busca(Tipo dado);
    void remove(ItTipo it);
};
```

Utilização da SacoVetor1

Antes de completar as funções pendentes, utilizaremos a `SacoVetor1`:

```
int main () {
   SacoTAD s = SacoVetor1();
   s.cria();
   s.adiciona('c');
   s.adiciona('a');
   s.adiciona('b');
   s.adiciona('a');
   printf("%d\n", s.busca('a').terminou());
   printf("%d\n", s.busca('d').terminou());
   for(auto it = s.itera(); !it.terminou(); it.proximo()) {
      printf("%c\n", it.atual()); // c a b a (nesta ordem)
   }
   l.libera();

   ...
```

Implementação: Cria e Libera

A operação `cria` inicializa a estrutura para uso, e a função `libera` não desaloca nada.

```
class SacoVetor1 {
...
void cria() {
    this->N = 0;
}

void libera() {
    this->N = 0;
}
...
}
```

Implementação: Adiciona

A operação `adiciona` no final do vetor.

```
class SacoVetor1 {
...
void adiciona(char dado) {
    this->elementos[N] = dado;
    this->N++; // N = N + 1
}
...
}
```

Implementação do Iterador

Precisamos ainda de uma definição de iterador.

```
class IteradorVetor1
{
public:
    char* elemento;  // elemento atual
    char* sentinela; // elemento sentinela "final"
    char atual() { return *this->elemento };
    bool terminou() { return elemento == sentinela; }
    void proximo() { this->elemento++; }
};
// verifica se agregado satisfaz conceito IteradorTAD
static_assert(IteradorTAD<IteradorVetor1>);
// verifica se agregado SacoVetor1 satisfaz conceito SacoTAD
static_assert(SacoTAD<SacoVetor1>);
```

Implementação: Itera

A operação itera adiciona a posição atual (vetor decai a um ponteiro), bem como uma posição sentinela final.

```
class SacoVetor1 {
...
IteradorVetor1 itera() {
    IteradorVetor1 it{
        .elemento = this->elementos,
        .sentinela = this->elementos + N};
    return it;
}
...
}
```

Implementação: Busca

A operação busca em um vetor faz um percurso direto (alternativa sem utilizar o iterador).

```
class SacoVetor1 {
...
IteradorVetor1 busca(char dado) {
   for (int i = 0; i < N; i++)
      if (elementos[i] == dado)
         return IteradorVetor1{
              .elemento = this->elementos + i,
              .sentinela = this->elementos + N};
   return IteradorVetor1{
        .elemento = this->elementos + N,
        .sentinela = this->elementos + N};
}
...
}
```

Implementação: Busca Recursiva

A busca pode ser feita de forma recursiva também (utilizando o iterador):

```
IteradorVetor1 buscarec(IteradorVetor1 it, char dado)
{
   if(it.terminou() || it.atual() == dado)
      return it;
   else
   {
      it.proximo();
      return buscarec(it, dado);
   }
}
```

Implementação: Remoção

A remoção de um elemento exige realocação/cópia dos elementos posteriores.

```
class SacoVetor1 {
...
void remove(IteradorVetor1 it)
{
    auto* i = it.elemento;
    while (i != (this->elementos + N)) {
        (*i) = *(i + 1);
        i++;
    }
    this->N--; // N = N - 1
}
...
}
```

Fim implementações

Fim parte de implementações.

Section 6

Análise de Complexidade

Saco: Revisão Geral

- Para que serve um saco?
- Quais são os métodos de uma saco?
- Qual é a complexidade de cada método de uma Lista Encadeada?
- Quais recursos são desejáveis para estender a funcionalidade de uma lista?

Bibliografia Recomendada

Além da bibliografia do curso, recomendamos para esse tópico:

- Szwarcfiter, J.L; Markenzon, L. Estruturas de Dados e seus Algoritmos. Rio de Janeiro, LTC, 1994. Bibliografia Adicional:
- Cerqueira, R.; Celes, W.; Rangel, J.L. Introdução a estruturas de dados: com técnicas de programação em C. Editora, 2004.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein Algoritmos: Teoria e Prática. Ed. Campus, 2002.
- Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Introduction to Algorithms, 3rd ed.. The MIT Press, 2009.
- Preiss, B.R. Estruturas de Dados e Algoritmos Ed. Campus, 2000;
- Knuth, D.E. The Art of Computer Programming - Vols I e III. 2nd Edition. Addison Wesley, 1973.
- Graham, R.L., Knuth, D.E., Patashnik, O. Matemática Concreta. Segunda Edição, Rio de Janeiro, LTC, 1995.
- Livro "The C++ Programming Language" de Bjarne Stroustrup
- Dicas e normas C++: https://github.com/isocpp/CppCoreGuidelines

Section 7

Agradecimentos

Pessoas

Em especial, agradeço aos colegas que elaboraram bons materiais, como o prof. Fabiano Oliveira (IME-UERJ), e o prof. Jayme Szwarcfiter cujos conceitos formam o cerne desses slides.

Estendo os agradecimentos aos demais colegas que colaboraram com a elaboração do material do curso de Pesquisa Operacional, que abriu caminho para verificação prática dessa tecnologia de slides.

Software

Esse material de curso só é possível graças aos inúmeros projetos de código-aberto que são necessários a ele, incluindo:

- pandoc
- LaTeX
- GNU/Linux
- git
- markdown-preview-enhanced (github)
- visual studio code
- atom
- revealjs
- groomit-mpx (screen drawing tool)
- xournal (screen drawing tool)
- ...

Empresas

Agradecimento especial a empresas que suportam projetos livres envolvidos nesse curso:

- github
- gitlab
- microsoft
- google
- ...

Reprodução do material

Esses slides foram escritos utilizando pandoc, segundo o tutorial ilectures:

- https://igormcoelho.github.io/ilectures-pandoc/

Exceto expressamente mencionado (com as devidas ressalvas ao material cedido por colegas), a licença será Creative Commons.

Igor Machado Coelho

This Slide Is Intentionally Blank (for goomit-mpx)