Listas Lineares

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará
Campus Quixadá

1º semestre/2020

Introdução

Introdução

- Uma estrutura de dados armazena dados na memória do computador a fim de permitir o acesso eficiente dos mesmos.

- A maioria das estruturas de dados usam como recurso principal a memória primária (a chamada RAM) como pilhas, filas, árvores binárias de busca, árvores AVL e árvores rubro-negras.

- Outras são especialmente projetadas e adequadas para serem armazenadas em memórias secundárias como o disco rígido, como as árvores B.

- Uma estrutura de dados bem projetada permite a manipulação eficiente, em tempo e em espaço, dos dados armazenados através de operações específicas.

Lista linear — Definição

- Uma lista linear L é um conjunto de $n \geq 0$ nós (ou células) $L_0, L_1, \ldots, L_{n-1}$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:
  - Se $n > 0$, L_0 é o primeiro nó,
  - Para $0 < k \leq n - 1$, o nó L_k é precedido por L_{k-1}.

Lista linear — Definição

- Uma lista linear L é um conjunto de $n \geq 0$ nós (ou células) $L_0, L_1, \ldots, L_{n-1}$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear:
  - Se $n > 0$, L_0 é o primeiro nó,
  - Para $0 < k \leq n - 1$, o nó L_k é precedido por L_{k-1}.
- Os nós de uma lista linear armazenam informações referentes a um conjunto de elementos que se relacionam entre si.
  - Informações sobre os funcionários de uma empresa.
  - Notas de alunos
  - Itens de estoque, etc.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.
- Combinar duas ou mais listas lineares em uma só.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.
- Combinar duas ou mais listas lineares em uma só.
- Quebrar uma lista linear em duas ou mais.

Lista linear – Operações

Algumas operações que podemos querer realizar sobre listas lineares:

- Ter acesso a L_k, $0 \leq k \leq n-1$ qualquer, a fim de examinar ou alterar o conteúdo de seus campos.
  - Se fixamos $k = n-1$, temos uma ED chamada Pilha.
  - Se fixamos $k = 0$, temos uma ED chamada Fila.
- Inserir um elemento novo antes ou depois de L_k.
- Remover L_k.
- Colocar todos os elementos da lista em ordem.
  - Estudaremos algoritmos de ordenação no final do curso.
- Combinar duas ou mais listas lineares em uma só.
- Quebrar uma lista linear em duas ou mais.
- Copiar uma lista linear em um outro espaço.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao L_k, para k qualquer.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao L_k, para k qualquer.
(2) inserir ou remover elementos em qualquer posição da lista linear.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao L_k, para k qualquer.
(2) inserir ou remover elementos em qualquer posição da lista linear.

A operação (1) fica eficiente se a lista é implementada em um vetor (array) em alocação sequencial na memória.

Implementação de uma lista linear

- O modo de implementar listas lineares depende da classe de operações mais frequentes. Não existe, em geral, uma única implementação para a qual todas as operações são eficientes.

- Por exemplo, não existe uma implementação para atender às seguintes duas operações de maneira eficiente:

(1) ter acesso fácil ao L_k, para k qualquer.
(2) inserir ou remover elementos em qualquer posição da lista linear.

A operação (1) fica eficiente se a lista é implementada em um vetor (array) em alocação sequencial na memória.

Para a operação (2) é mais adequada a alocação encadeada, com o uso de ponteiros.

Tipos de alocação

O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa na memória de dois nós consecutivos na lista.

- **Alocação sequencial:** dois nós consecutivos na lista estão em posições contíguas de memória.

- **Alocação encadeada:** dois nós consecutivos na lista podem estar em posições não contíguas da memória.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Listas Sequenciais

Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.

L	L+c	L+2c	L+3c	L+4c	L+5c	L+6c
nó 1	nó 2	nó 3	nó 4	nó 5	nó 6	nó 7

Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.

L	L+c	L+2c	L+3c	L+4c	L+5c	L+6c
nó 1	nó 2	nó 3	nó 4	nó 5	nó 6	nó 7

- Neste caso, o endereço real do $(j+1)$-ésimo nó da lista se encontra c unidades adiante daquele correspondente ao j-ésimo. A constante c é o número de bytes que cada nó ocupa.

Listas sequenciais (Vetores)

- Nós em posições contíguas da memória.

L	L+c	L+2c	L+3c	L+4c	L+5c	L+6c
nó 1	nó 2	nó 3	nó 4	nó 5	nó 6	nó 7

- Neste caso, o endereço real do $(j+1)$-ésimo nó da lista se encontra c unidades adiante daquele correspondente ao j-ésimo. A constante c é o número de bytes que cada nó ocupa.

- A correspondência entre o índice da lista e o endereço real é feita automaticamente pela linguagem de programação quando da compilação do programa.

TAD Lista Sequencial

- Lista sequencial pode ser modelada como um Tipo Abstrato de Dados.

TAD Lista Sequencial

- Lista sequencial pode ser modelada como um Tipo Abstrato de Dados.

- O TAD Lista Sequencial tem os seguintes atributos:
  - um vetor de inteiros.
  - a capacidade total do vetor.
  - a quantidade de elementos no vetor.

TAD Lista Sequencial

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Lista sequencial pode ser modelada como um Tipo Abstrato de Dados.

- O TAD Lista Sequencial tem os seguintes atributos:
  - um vetor de inteiros.
  - a capacidade total do vetor.
  - a quantidade de elementos no vetor.

- Operações possíveis são:
  - Criar lista .
  - Liberar lista.
  - Consultar o tamanho atual da lista.
  - Saber se lista está cheia.
  - Buscar um elemento e retornar sua posição na lista.
  - Adicionar um elemento ao final da lista.
  - Remover um elemento da lista.

Implementação usando Classe

Arquivo SeqList.h


```
1  #ifndef SEQLIST_H
2  #define SEQLIST_H
3
4  class SeqList {
5      private:
6          int *vec = nullptr; // Vetor de inteiros
7          int size_vec = 0; // Qtd de elementos no vetor
8          int capacity_vec = 0; // Capacidade total do vetor
9      public:
10         SeqList(int n); // Construtor: recebe capacidade
11         ~SeqList(); // Destrutor: libera memoria alocada
12         bool add(int x); // Adiciona x ao final da lista
13         void remove(int x); // Remove o primeiro x da lista
14         int search(int x); // Busca x e retorna indice
15         int at(int k); // Retorna o k-esimo elemento da lista
16         int size(); // Retorna tamanho do SeqList
17         bool isFull(); // Retorna se lista esta cheia
18         void clear(); // Deixa a lista vazia
19         void print(); // Imprime elementos
20 };
21
22 #endif
```

Arquivo SeqList.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Implementacao da classe SeqList
2  #include <iostream>
3  #include "SeqList.h"
4
5  // Construtor
6  SeqList::SeqList(int n) {
7      if(n > 0) {
8          vec = new int[n];
9          capacity_vec = n;
10         size_vec = 0;
11     }
12 }
13
14 // Destrutor
15 SeqList::~SeqList() {
16     if(vec != nullptr)
17         delete[] vec;
18 }
```

Continuação do arquivo SeqList.cpp


```
1  // Retorna numero de elementos na lista
2  int SeqList::size() {
3      return size_vec;
4  }
5
6  // Retorna se a lista esta cheia ou nao
7  bool SeqList::isFull() {
8      return size_vec == capacity_vec;
9  }
10
11 // Recebe um inteiro x como argumento e adiciona o
12 // inteiro x logo apos o ultimo elemento da lista.
13 // Retorna 'true' se for bem sucedido, ou 'false'
14 // caso contrario. Nenhum elemento deve ser adicionado
15 // se a lista estiver cheia.
16 bool SeqList::add(int x) {
17     if(isFull())
18         return false;
19     vec[size_vec] = x;
20     size_vec++;
21     return true;
22 }
```

Continuação do arquivo SeqList.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Busca um elemento x e retorna seu indice se
2  // ele existir, ou -1 caso contrario
3  int SeqList::search(int x) {
4      for(int i = 0; i < size_vec; i++)
5          if(vec[i] == x)
6              return i;
7      return -1;
8  }
9
10 // Retorna o elemento no indice 'index'
11 // Supoe que o indice passado eh valido
12 int SeqList::at(int index) {
13     return vec[index];
14 }
15
16 // Apos essa operacao, a lista fica vazia.
17 void SeqList::clear() {
18     size_vec = 0;
19 }
```

Final do arquivo SeqList.cpp

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```
1  // Imprime todos os elementos da lista em uma unica linha.
2  void SeqList::print() {
3      for(int i = 0; i < size_vec; i++)
4          std::cout << vec[i] << " ";
5  }
6
7  // Remove a primeira ocorrencia do elemento x na SeqList
8  void SeqList::remove(int x) {
9      int index = search(x);
10     if(index != -1) {
11         while(index <= size_vec-2) {
12             vec[index] = vec[index+1];
13             index++;
14         }
15         size_vec--;
16     }
17 }
```

Programa cliente main.cpp


```
1  #include <iostream>
2  #include "SeqList.h"
3  #define MAX 10
4  using namespace std;
5
6  int main() {
7      SeqList lista(MAX); // Criando lista sequencial
8
9      int i = 1;
10     while(!lista.isFull()) { lista.add(i++); }
11
12     for(i = 0; i < lista.size(); i++)
13         cout << lista.at(i) << " ";
14
15     lista.remove(5);
16     lista.remove(3);
17
18     cout << "\nLista com 3 e 5 removidos: " << endl;
19     lista.print();
20     cout << endl;
21     return 0;
22 }
```

Exercícios

Exercícios

Implemente as seguintes operações adicionais na Lista Sequencial:

- `bool replaceAt(int x, int k)`: Troca o elemento na posicao k pelo elemento x (somente se $0 \leq k \leq size\_vec - 1$)

- `void removeAt(int k)`: Remove o elemento com índice k na lista. Deve-se ter $0 \leq$ `k` $\leq$ `size_vex-1`; caso contrario, a remoção não é realizada.

- `bool insertAt(int x, int k)`: Adiciona o elemento x na posicao k (somente se $0 \leq k \leq size\_vec$ e $size\_vec < max\_vec$). Antes de fazer a inserção, todos os elementos da posição k em diante são deslocados uma posição para a direita.

- `void removeAll(int x)`: Remove todas as ocorrências do elemento x na lista.

Listas Encadeadas

Vetores (algumas considerações)

Vetores:

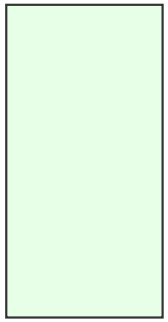
- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado

Vetores (algumas considerações)

Vetores:

- estão alocados contiguamente na memória
  - pode ser que tenhamos espaço na memória
  - mas não para alocar um vetor do tamanho desejado
- tem um tamanho fixo
  - ou alocamos um vetor pequeno e o espaço pode acabar
  - ou alocamos um vetor grande e desperdiçamos memória

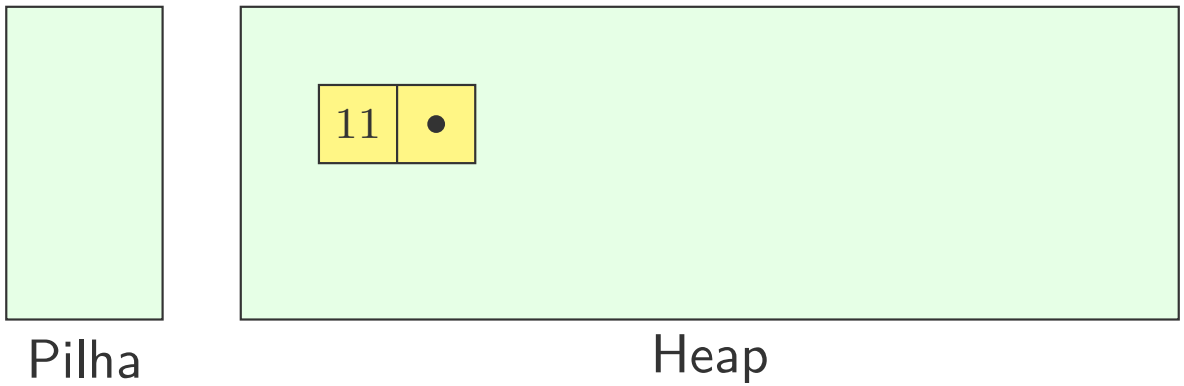
Alternativa - Lista Simplesmente Encadeada



Pilha

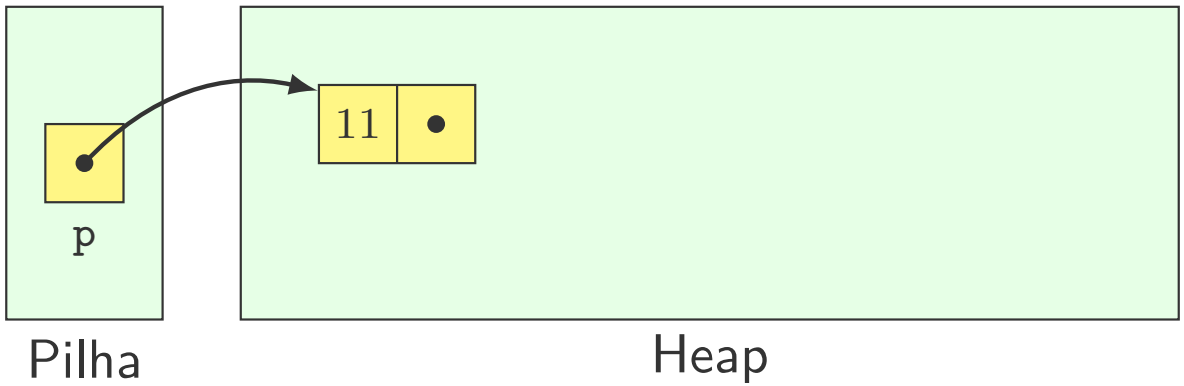
Alternativa - Lista Simplesmente Encadeada

Alternativa - Lista Simplesmente Encadeada



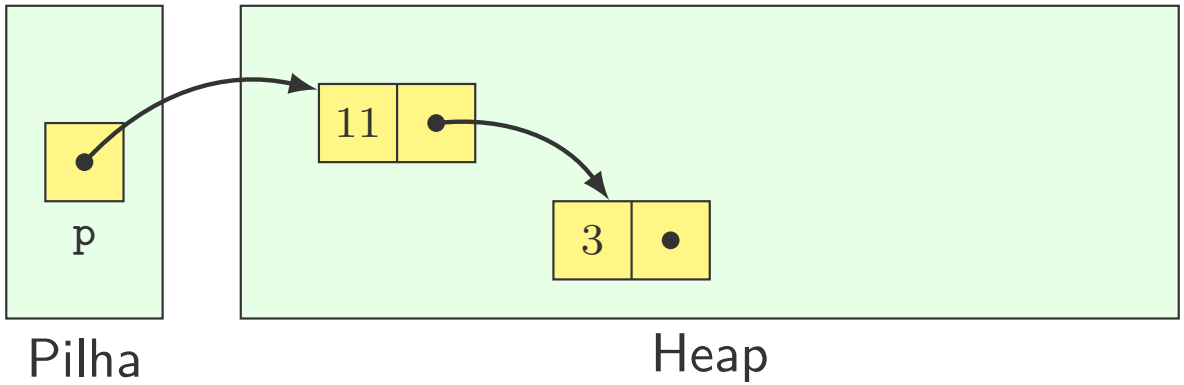
- alocamos memória conforme o necessário

Alternativa - Lista Simplesmente Encadeada



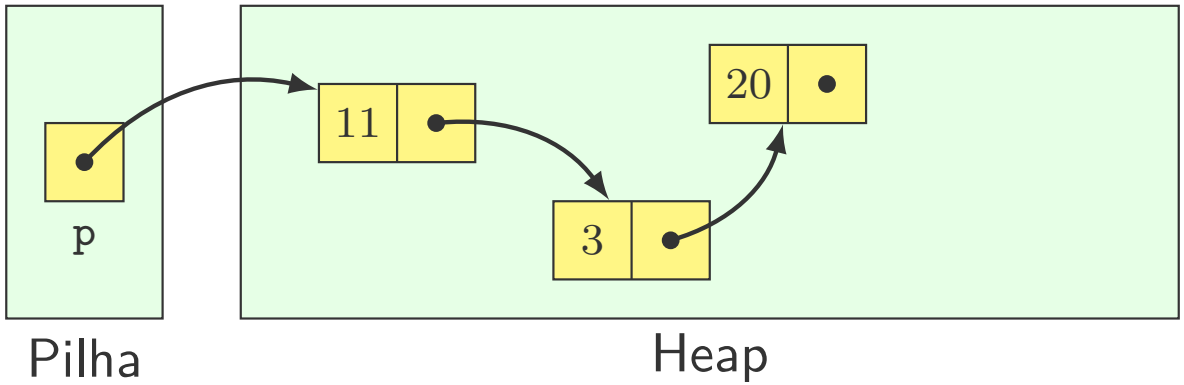
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável

Alternativa - Lista Simplesmente Encadeada



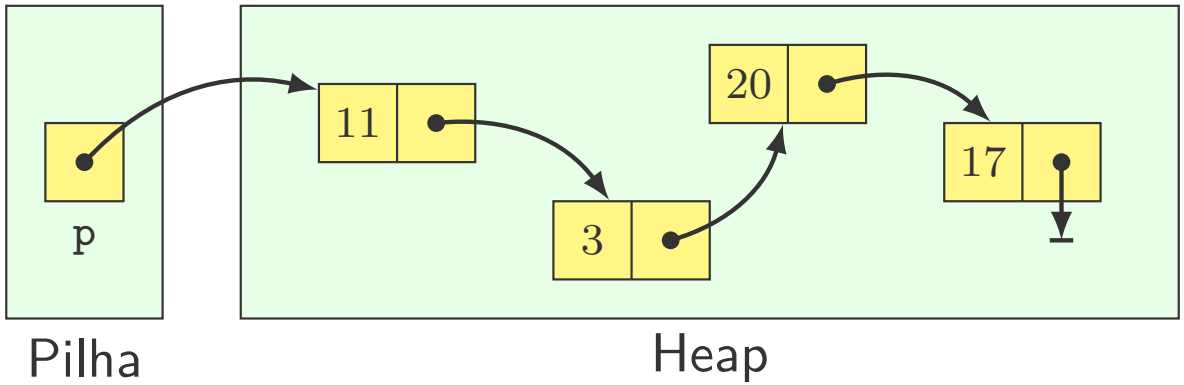
- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo

Alternativa - Lista Simplesmente Encadeada



- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro

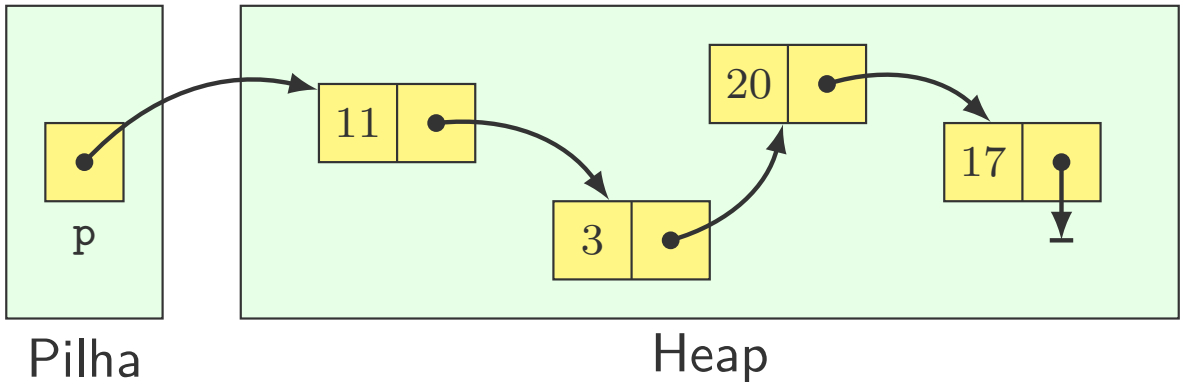
Alternativa - Lista Simplesmente Encadeada



- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro

Alternativa - Lista Simplesmente Encadeada

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ



- alocamos memória conforme o necessário
- guardamos um ponteiro para a estrutura em uma variável
- o primeiro nó aponta para o segundo
- o segundo nó aponta para o terceiro
- o último nó aponta para `nullptr`

Listas Simplesmente Encadeadas

Nó: elemento alocado dinamicamente que contém:

- um conjunto de dados
- um ponteiro para outro nó

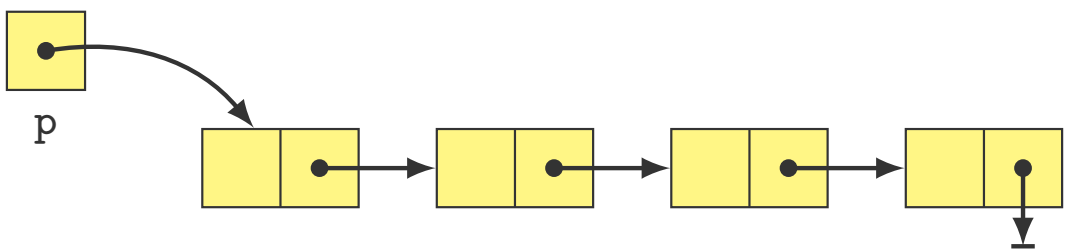
Listas Simplesmente Encadeadas

Nó: elemento alocado dinamicamente que contém:

- um conjunto de dados
- um ponteiro para outro nó

Lista encadeada:

- Conjunto de nós ligados entre si de maneira sequencial



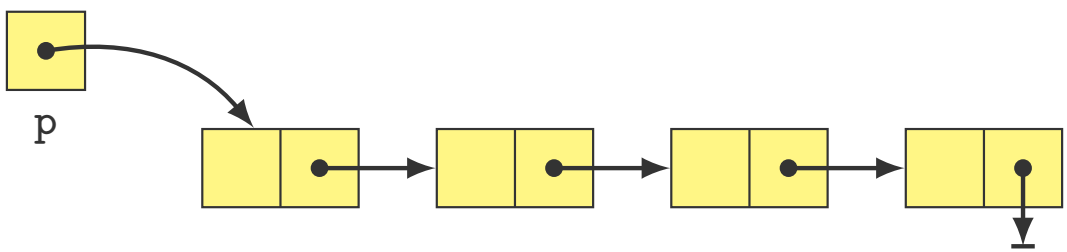
Listas Simplesmente Encadeadas

Nó: elemento alocado dinamicamente que contém:

- um conjunto de dados
- um ponteiro para outro nó

Lista encadeada:

- Conjunto de nós ligados entre si de maneira sequencial



Observações:

- a lista encadeada é acessada a partir de um ponteiro
- um ponteiro pode estar vazio (aponta para `nullptr` em C++)

Implementação da Lista Encadeada

Definição do nó


```
1 struct Node {
2     /* valor a ser guardado */
3     int value;
4
5     /* ponteiro para o proximo no da lista */
6     Node *next;
7 };
```

- Implementamos cada nó que compõe a lista simplesmente encadeada como uma estrutura que contém dois campos.

Definição do nó


```
1 struct Node {
2     /* valor a ser guardado */
3     int value;
4
5     /* ponteiro para o proximo no da lista */
6     Node *next;
7 };
```

- Implementamos cada nó que compõe a lista simplesmente encadeada como uma estrutura que contém dois campos.

- A lista tem um nó especial, chamado nó cabeça. Esse nó servirá apenas para marcar o início da lista. Seu valor armazenado não possui significado.

Arquivo List.h


```
1  #ifndef LIST_H
2  #define LIST_H
3
4  struct Node;
5
6  class List {
7     private:
8        Node *head; // Ponteiro para a cabeca da lista
9        // Operacao auxiliar: retorna no antecessor do no
10       // com valor x, ou NULL caso x nao esteja presente
11       Node *search(int x);
12    public:
13       List(); // Construtor
14       ~List(); // Destrutor: libera memoria alocada
15       void add(int x); // Insere x ao final da lista
16       void remove(int x); // remove o primeiro no com valor x
17       void removeAll(int x); // remove todo no com valor x
18       int removeNodeAt(int k); // remove k-esimo no
19       void print(); // Imprime elementos
20       bool isEmpty(); // Esta vazia?
21       int size(); // retorna numero de nos
22       void clear(); // deixa a lista vazia
23 };
24 #endif
```

Continuação do Arquivo `List.cpp`

Definição do nó:

Continuação do Arquivo `List.cpp`

Definição do nó:

```
1  #include <iostream>
2  #include "List.h"
3
4  struct Node {
5      /* valor a ser guardado */
6      int value;
7
8      /* ponteiro para o proximo no da lista */
9      Node *next;
10 };
```

Continuação do Arquivo `List.cpp`

Criando uma lista vazia:

Continuação do Arquivo `List.cpp`

Criando uma lista vazia:

```
1 List::List() { // Construtor
2     head = new Node;
3     head->value = 0;
4     head->next = NULL;
5 }
```

Continuação do Arquivo `List.cpp`

Criando uma lista vazia:

```
1 List::List() { // Construtor
2     head = new Node;
3     head->value = 0;
4     head->next = NULL;
5 }
```

Limpando a lista (recursivamente):

Continuação do Arquivo `List.cpp`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Criando uma lista vazia:

```
1  List::List() { // Construtor
2      head = new Node;
3      head->value = 0;
4      head->next = NULL;
5  }
```

Limpando a lista (recursivamente):

```
1  void List::clear() {
2      clearRecursive(head->next);
3      head->next = NULL;
4  }
5
6  void List::clearRecursive(Node *node) {
7      if(node != NULL) {
8          clearRecursive(node->next);
9          delete node;
10     }
11 }
```

Continuação do Arquivo `List.cpp`

Criando uma lista vazia:

```
1 List::List() { // Construtor
2     head = new Node;
3     head->value = 0;
4     head->next = NULL;
5 }
```

Limpando a lista (recursivamente):

```
1 void List::clear() {
2     clearRecursive(head->next);
3     head->next = NULL;
4 }
5
6 void List::clearRecursive(Node *node) {
7     if(node != NULL) {
8         clearRecursive(node->next);
9         delete node;
10    }
11 }
```

Exercício: Faça uma versão iterativa de `clear()`

Continuação do Arquivo `List.cpp`

Destrutor:

Continuação do Arquivo `List.cpp`

Destrutor:

```
1 List::~List() {
2     clear();
3     delete head;
4 }
```

Continuação do Arquivo `List.cpp`

Destrutor:

```
1 List::~List() {
2     clear();
3     delete head;
4 }
```

Inserção:

Continuação do Arquivo `List.cpp`

Destrutor:

```
1 List::~List() {
2     clear();
3     delete head;
4 }
```

Inserção:

```
1 void List::add(int x) {
2     Node *ant = head;
3     while(ant->next != NULL)
4         ant = ant->next;
5     Node *novo = new Node;
6     novo->value = x;
7     novo->next = NULL;
8     ant->next = novo;
9 }
```

Continuação do Arquivo `List.cpp`

Destrutor:

```
1 List::~List() {
2     clear();
3     delete head;
4 }
```

Inserção:

```
1 void List::add(int x) {
2     Node *ant = head;
3     while(ant->next != NULL)
4         ant = ant->next;
5     Node *novo = new Node;
6     novo->value = x;
7     novo->next = NULL;
8     ant->next = novo;
9 }
```

- Qual a complexidade da inserção?

Continuação do Arquivo `List.cpp`

Destrutor:

```
1 List::~List() {
2     clear();
3     delete head;
4 }
```

Inserção:

```
1 void List::add(int x) {
2     Node *ant = head;
3     while(ant->next != NULL)
4         ant = ant->next;
5     Node *novo = new Node;
6     novo->value = x;
7     novo->next = NULL;
8     ant->next = novo;
9 }
```

- Qual a complexidade da inserção?
- A inserção ocorre em $O(n)$

Continuação do Arquivo `List.cpp`

Destrutor:

```
1 List::~List() {
2     clear();
3     delete head;
4 }
```

Inserção:

```
1 void List::add(int x) {
2     Node *ant = head;
3     while(ant->next != NULL)
4         ant = ant->next;
5     Node *novo = new Node;
6     novo->value = x;
7     novo->next = NULL;
8     ant->next = novo;
9 }
```

- Qual a complexidade da inserção?
- A inserção ocorre em $O(n)$
- É possível fazer melhor? Quais as implicações?

Continuação do arquivo `List.cpp`

Impressão iterativa:

Continuação do arquivo `List.cpp`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Impressão iterativa:

```
1 void List::print() {
2     Node *aux = head->next;
3     while(aux != NULL) {
4         std::cout << aux->value << " ";
5         aux = aux->next;
6     }
7 }
```

Continuação do arquivo `List.cpp`

Impressão iterativa:

```
1 void List::print() {
2     Node *aux = head->next;
3     while(aux != NULL) {
4         std::cout << aux->value << " ";
5         aux = aux->next;
6     }
7 }
```

Impressão recursiva:

Continuação do arquivo `List.cpp`

Impressão iterativa:

```
1  void List::print() {
2      Node *aux = head->next;
3      while(aux != NULL) {
4          std::cout << aux->value << " ";
5          aux = aux->next;
6      }
7  }
```

Impressão recursiva:

```
1  void List::printRecursive() {
2      printRecursive(head->next);
3  }
4
5  void List::printRecursive(Node *node) {
6      if (node != NULL) {
7          std::cout << node->value << " ";
8          printRecursive(node->next);
9      }
10 }
```

Final do arquivo `List.cpp`

Remoção da primeira ocorrência de um elemento

Final do arquivo `List.cpp`

Remoção da primeira ocorrência de um elemento

Vamos precisar da seguinte função auxiliar:

Final do arquivo `List.cpp`

Remoção da primeira ocorrência de um elemento

Vamos precisar da seguinte função auxiliar:

```
1 // Retorna ponteiro para o no anterior ao do elemento x,
2 // ou retorna NULL caso x nao esteja presente
3 Node* List::search(int x) {
4     Node *node = head;
5     while(node->next != NULL && (node->next)->value != x)
6         node = node->next;
7     return (node->next == NULL) ? NULL : node;
8 }
```

Final do arquivo `List.cpp`

Remoção da primeira ocorrência de um elemento

Vamos precisar da seguinte função auxiliar:

```
1  // Retorna ponteiro para o no anterior ao do elemento x,
2  // ou retorna NULL caso x nao esteja presente
3  Node* List::search(int x) {
4      Node *node = head;
5      while(node->next != NULL && (node->next)->value != x)
6          node = node->next;
7      return (node->next == NULL) ? NULL : node;
8  }
```

```
1  void List::remove(int x) {
2      Node *anterior = search(x);
3      if(anterior != NULL) {
4          Node *aux;
5          aux = anterior->next;
6          anterior->next = aux->next;
7          aux->next = NULL;
8          delete aux;
9      }
10 }
```

Programa cliente `main.cpp`


```
1  #include <iostream>
2  #include "List.h"
3  using namespace std;
4
5  int main() {
6      List lista;
7
8      for(int i = 1; i <= 15; i++)
9          lista.add(i);
10
11     lista.print();
12     lista.remove(5);
13     lista.print();
14
15     return 0;
16 }
```

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$
- Remoção da posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a esquerda)
  - Lista: $O(1)$

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$
- Remoção da posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a esquerda)
  - Lista: $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Listas sequenciais $\times$ Listas encadeadas

- Acesso à posição k:
  - Vetor: $O(1)$
  - Lista: $O(k)$ (precisa percorrer a lista)
- Inserção na posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a direita)
  - Lista: $O(1)$
- Remoção da posição 0:
  - Vetor: $O(n)$ (precisa mover itens para a esquerda)
  - Lista: $O(1)$
- Uso de espaço:
  - Vetor: provavelmente desperdiçará memória
  - Lista: não desperdiça memória, mas cada elemento consome mais memória por causa do ponteiro

Qual é melhor?

- depende do problema, do algoritmo e da implementação

Exercício 1

Outras operações em lista encadeada

Exercícios: Implemente as demais operações sobre listas:

- `bool isEmpty()`: Retorna se lista está ou não vazia
- `int size()`: Retorna número de nós.
- `void removeAll(int x)`: Remove da lista todas as ocorrências do inteiro x.
- `int removeNodeAt(int k)`: Remove a k-ésima célula da lista encadeada e retorna o seu valor. Note que deve-se ter $1 \leq$ `k` $\leq$ `size()`; caso contrário, retorna-se o menor inteiro possível.

Outras operações em lista encadeada

- `void insertAfter(int x, int k)`: Insere um novo nó com valor x após o k-ésimo nó da lista.
  Deve-se ter $0 \leq$ `k` $\leq$ `size()` para que a inserção seja realizada; caso contrário, não sera realizada.

- `QX_List *copy()`: Retorna um ponteiro para uma cópia desta lista.

- `void copyArray(int v[], int n)`: Copia os elementos do array `v` para a lista. O array tem n elementos. Todos os elementos anteriores da lista são apagados.

Outras operações em lista encadeada

- `bool equal(QX_List *lst)`: Determina se a lista `lst`, passada por parâmetro, é igual a lista em questao. Duas listas são iguais se elas tem o mesmo tamanho e o valor do k-ésimo elemento da primeira lista é igual ao k-ésimo valor da segunda lista.

- `void concat(QX_List *lst)`: Concatena a lista atual com a lista `lst` passada por referência. Após essa operação, `lst` será uma lista vazia, ou seja, o único nó de `lst` sera o nó cabeça.

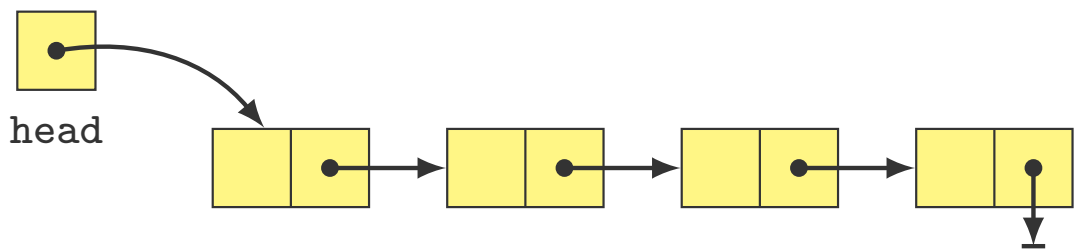
- `void reverse()`: Inverte a ordem dos nós (o primeiro nó passa a ser o último, o segundo passa a ser o penúltimo, etc.) Essa operação faz isso sem criar novos nós, apenas altera os ponteiros.

Exercício 2

Implementação de lista encadeada sem nó cabeça

Exercício: Implemente a lista simplesmente encadeada sem usar nó cabeça. Assim como a lista que vimos nesta aula, essa nova lista deve ter apenas um atributo, que é o ponteiro `head` do tipo `Node`.

- Quando a lista estiver vazia, `head` aponta para `NULL`.
- Caso contrário, `head` deve apontar diretamente para o primeiro nó da lista.



- Implemente todas as operações vistas nesta aula.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM