45EST – Algoritmos e Estruturas de Dados

Listas dinâmicas

Prof. Marcelo de Souza

UDESC Ibirama
Bacharelado em Engenharia de Software

marcelo.desouza@udesc.br
Versão compilada em 1 de setembro de 2018

Leitura obrigatória:

- Capítulo 7 de Goodrich et al. [2014] – Listas e iteradores.

Leitura complementar:

- Capítulo 5 de Lafore and Machado [2004] – Listas encadeadas.
- Capítulo 7 de Pereira [2008] – Listas encadeadas.

Listas

Pilhas, filas e deques:

- Eficientes. ← operações $O(1)$.
- Permitem inserção/remoção nas extremidades.

Lista dinâmica: estrutura que permite operações em posições arbitrárias.

Principais métodos:

- `get(i)`: retorna o i-ésimo elemento da lista.
- `set(i, e)`: substitui o i-ésimo elemento por e.
- `add(i, e)`: adiciona e na posição i e desloca os demais elementos.
- `remove(i)`: remove o i-ésimo elemento da lista.

Exemplo de funcionamento:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	“error”	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	“error”	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	“error”	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

Interface `List`:

```
1  public interface List<E> {
2     int size();
3     boolean isEmpty();
4     E get(int i);
5     E set(int i, E e);
6     void add(int i, E e);
7     E remove(int i);
8  }
```

Implementação com vetores:

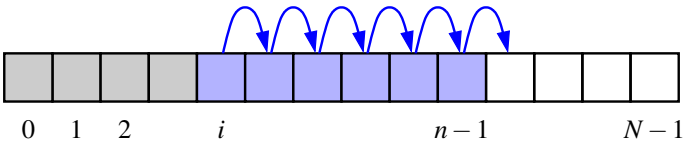
- Elementos precisam ser realocados.
- Tamanho estático da estrutura.

Implementação com encadeamento:

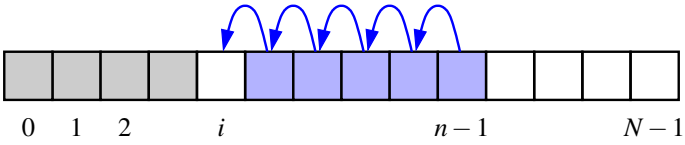
- Operação interna implica em percorrer a lista.

Realocação dos elementos de um vetor:

- Na inserção, todos os elementos são movidos para trás.

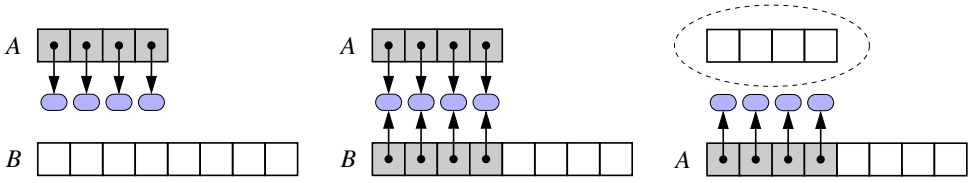


- Na remoção, todos os elementos são movidos para frente.



Contornando a estrutura fixa de um vetor:

- Mantém uma lista de tamanho fixo.
- Quando precisar de mais espaço, dobra o tamanho da lista.
  - Cria uma nova (maior) lista.
  - Copia os elementos para a nova lista.
  - Atualiza a referência.



Implementação baseada em vetores:

```
1  public class ArrayList<E> implements List<E> {
2     public static final int CAPACITY=16;
3     private E[] data;
4     private int size = 0;
5
6     public ArrayList() { this(CAPACITY); }
7
8     public ArrayList(int capacity) {
9        data = (E[]) new Object[capacity];
10    }
11
12    public int size() { return size; }
13    public boolean isEmpty() { return size == 0; }
14
15    public E get(int i) {
16       checkIndex(i, size);
17       return data[i];
18    }
19
20    public E set(int i, E e) {
21       checkIndex(i, size);
22       E temp = data[i];
23       data[i] = e;
24       return temp;
25    }
26
27    public void add(int i, E e) {
28       checkIndex(i, size + 1);
29       if (size == data.length)
30          resize(2 * data.length);
31       for (int k=size-1; k >= i; k--)
32          data[k+1] = data[k];
33       data[i] = e;
34       size++;
35    }
36
```

```
37      public E remove(int i) {
38         checkIndex(i, size);
39         E temp = data[i];
40         for (int k=i; k < size-1; k++)
41            data[k] = data[k+1];
42         data[size-1] = null;
43         size--;
44         return temp;
45      }
46
47      protected void checkIndex(int i, int n) {
48         if (i < 0 || i >= n)
49            throw new IndexOutOfBoundsException("Illegal index: " + i);
50      }
51
52      protected void resize(int capacity) {
53         E[] temp = (E[]) new Object[capacity];
54         for (int k=0; k < size; k++)
55            temp[k] = data[k];
56         data = temp;
57      }
58
59      public String toString() {
60         StringBuilder sb = new StringBuilder("(");
61         for (int j = 0; j < size; j++) {
62            if (j > 0) sb.append(", ");
63            sb.append(data[j]);
64         }
65         sb.append(")");
66         return sb.toString();
67      }
68   }
```

Comentários:

- O método `checkIndex` verifica se não será acessada uma posição fora da lista.
- Na adição, caso se atinja o limite de tamanho do vetor, é chamado o mé-

todo `resize` para aumentar sua capacidade e, então, inserir o elemento.

- A adição implica em movimentar todos os elementos seguintes.
- A remoção implica em movimentar todos os elementos seguintes.

Implementação baseada em listas encadeadas

Criação de métodos de acesso aleatório na lista duplamente encadeada:

```
1  public class DoublyLinkedList<E> {
2
3      /* ...
4      * Complete implementation
5      */
6
7      public void add(int position, E e) {
8          checkIndex(position, size + 1);
9          Node<E> n = searchNode(position);
10         addBetween(e, n.getPrev(), n);
11     }
12
13     public E get(int position) {
14         checkIndex(position, size);
15         return(searchNode(position).getElement());
16     }
17
18     public E set(int position, E e) {
19         checkIndex(position, size);
20         Node<E> n = searchNode(position);
21         addBetween(e, n, n.getNext());
22         remove(n);
23         return n.getElement();
24     }
25
26     public E remove(int position) {
27         checkIndex(position, size);
28         return remove(searchNode(position));
```

```
29      }
30
31      protected Node<E> searchNode(int position) {
32          if(position == 0) return header.getNext();
33          if(position == size()) return trailer;
34
35          int count = -1;
36          Node<E> walk = header.getNext();
37          while(walk != trailer) {
38              count++;
39              if(count == position) {
40                  return walk;
41              }
42              walk = walk.getNext();
43          }
44          return null;
45      }
46
47      protected void checkIndex(int i, int n) {
48          if (i < 0 || i >= n)
49              throw new IndexOutOfBoundsException("Illegal index: " + i);
50      }
51  }
```

Comentários:

- Antes de cada operação é verificada a posição de acesso.
- O método `searchNode` recupera o nodo da posição i.
- Os métodos básicos são usados após encontrar a posição.

Implementação da classe `LinkedList`:

```
1  public class LinkedList<E> implements List<E> {
2      DoublyLinkedList<E> list = new DoublyLinkedList<>();
3      public int size() { return list.size(); }
4      public boolean isEmpty() { return list.isEmpty(); }
```

```
5       public String toString() { return list.toString(); }
6       public E get(int i) { return list.get(i); }
7       public E set(int i, E e) { return list.set(i, e); }
8       public void add(int i, E e) { list.add(i, e); }
9       public E remove(int i) { return list.remove(i); }
10  }
```

Comparação de complexidade:

Operação	`ArrayList`	`LinkedList`
`get(i)`	$O(1)$	$O(n)$
`set(i,e)`	$O(1)$	$O(n)$
`add(i,e)`	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$

Exercícios:

1. Implemente listas dinâmicas baseadas em vetores e encadeamento para armazenar dados de filmes de uma locadora (título, gênero e ano). Implemente operações de inserção, remoção, consulta e listagem de filmes.

Listas posicionais

Ideia geral:

- Resolver a ineficiência da `LinkedList`.
- Outras formas para determinar a posição de um elemento:
  - Acessar diretamente os nodos → perde-se encapsulamento.
  - Fornecer uma interface para acesso aos nodos → posições.
- A posição de um elemento nunca muda, diferente do uso de índices.
- Exemplo: editor de texto (caracteres e cursor).

Interface `Position`:

```
1 public interface Position<E> {
2    E getElement();
3 }
```

TAD lista posicional:

- Métodos de acesso em função de uma posição:
  - `first()`/`last()` retornam a **posição** dos elementos.
  - `before(p)`/`after(p)` retornam **posições** antes/depois de outra.
  - Para acessar o elemento → `first().getElement()`.
- Métodos de atualização:
  - `addFirst(e)`/`addLast(e)`
  - `addBefore(p, e)`/`addAfter(p, e)`
  - `set(p, e)`/`remove(p)`.

Exemplo de funcionamento:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last( ))	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Interface `PositionalList`:

```
1  public interface PositionalList<E> {
2     int size();
3     boolean isEmpty();
4     Position<E> first();
5     Position<E> last();
6     Position<E> before(Position<E> p);
7     Position<E> after(Position<E> p);
8     Position<E> addFirst(E e);
9     Position<E> addLast(E e);
10    Position<E> addBefore(Position<E> p, E e);
11    Position<E> addAfter(Position<E> p, E e);
12    E set(Position<E> p, E e);
13    E remove(Position<E> p);
14 }
```

Implementação baseada em encadeadamento:

```
1  public class LinkedPositionalList<E> implements PositionalList<E> {
2
3     private static class Node<E> implements Position<E> {
4        private E element;
5        private Node<E> prev;
6        private Node<E> next;
7
8        public Node(E e, Node<E> p, Node<E> n) {
9           element = e;
10          prev = p;
11          next = n;
12       }
13
14       public E getElement() {
15          if (next == null)
16             throw new IllegalStateException("Pos. no longer valid");
17          return element;
18       }
19
20       public Node<E> getPrev() {
21          return prev;
22       }
23
24       public Node<E> getNext() {
25          return next;
26       }
27
28       public void setElement(E e) {
29          element = e;
30       }
31
32       public void setPrev(Node<E> p) {
33          prev = p;
34       }
35
36       public void setNext(Node<E> n) {
```

```
37            next = n;
38         }
39      }
40
41      private Node<E> header;
42      private Node<E> trailer;
43      private int size = 0;
44
45      public LinkedPositionalList() {
46         header = new Node<>(null, null, null);
47         trailer = new Node<>(null, header, null);
48         header.setNext(trailer);
49      }
50
51      private Node<E> validate(Position<E> p) {
52         if (!(p instanceof Node))
53            throw new IllegalArgumentException("Invalid p");
54         Node<E> node = (Node<E>) p;
55         if (node.getNext() == null)
56            throw new IllegalArgumentException("p is not in the list");
57         return node;
58      }
59
60      private Position<E> position(Node<E> node) {
61         if (node == header || node == trailer)
62            return null;
63         return node;
64      }
65
66      public int size() {
67         return size;
68      }
69
70      public boolean isEmpty() {
71         return size == 0;
72      }
73
74      public Position<E> first() {
```

```
75          return position(header.getNext());
76      }
77
78      public Position<E> last() {
79          return position(trailer.getPrev());
80      }
81
82      public Position<E> before(Position<E> p) {
83          Node<E> node = validate(p);
84          return position(node.getPrev());
85      }
86
87      public Position<E> after(Position<E> p) {
88          Node<E> node = validate(p);
89          return position(node.getNext());
90      }
91
92      private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
93          Node<E> newest = new Node<>(e, pred, succ);
94          pred.setNext(newest);
95          succ.setPrev(newest);
96          size++;
97          return newest;
98      }
99
100     public Position<E> addFirst(E e) {
101         return addBetween(e, header, header.getNext());
102     }
103
104     public Position<E> addLast(E e) {
105         return addBetween(e, trailer.getPrev(), trailer);
106     }
107
108     public Position<E> addBefore(Position<E> p, E e) {
109         Node<E> node = validate(p);
110         return addBetween(e, node.getPrev(), node);
111     }
112
```

```
113     public Position<E> addAfter(Position<E> p, E e) {
114        Node<E> node = validate(p);
115        return addBetween(e, node, node.getNext());
116     }
117
118     public E set(Position<E> p, E e) {
119        Node<E> node = validate(p);
120        E answer = node.getElement();
121        node.setElement(e);
122        return answer;
123     }
124
125     public E remove(Position<E> p) {
126        Node<E> node = validate(p);
127        Node<E> predecessor = node.getPrev();
128        Node<E> successor = node.getNext();
129        predecessor.setNext(successor);
130        successor.setPrev(predecessor);
131        size--;
132        E answer = node.getElement();
133        node.setElement(null);
134        node.setNext(null);
135        node.setPrev(null);
136        return answer;
137     }
138
139     public String toString() {
140        StringBuilder sb = new StringBuilder("(");
141        Node<E> walk = header.getNext();
142        while (walk != trailer) {
143           sb.append(walk.getElement());
144           walk = walk.getNext();
145           if (walk != trailer)
146           sb.append(", ");
147        }
148        sb.append(")");
149        return sb.toString();
150     }
```

```
151 }
```

Comentários:

- Implementação baseada em uma lista duplamente encadeada.
- Os nodos são as posições, mas o único aspecto visível externamente é o método `getElement`.
- Método `validate` faz a conversão de posição para nodo.
- Método `position` devolve o nodo convertido em posição.
- Uma posição inexistente é identificada pelos seus campos nulos.

OBS: uma vez conhecida a posição, todas as operações de uma lista posicional executam em tempo constante $O(1)$.

Atividades

1. Modifique o código das classes `ArrayList` e `LinkedList` e implemente um método `add(e)` que insere o elemento e no final da lista.
2. A complexidade dos métodos utilizados pela `LinkedList` origina-se no procedimento `searchNode`, reponsável por buscar o nodo da posição que se deseja acessar. O desempenho dessa estrutura de dados pode ser melhorado inibindo essa busca quando tratar-se de acesso ao início ou fim da lista. Implemente essa estratégia.
3. Outra forma de melhorar o desempenho de uma `LinkedList` é fazer com que a busca implementada no procedimento `searchNode` seja feita de "trás para frente", quando conveniente. Implemente essa estratégia. Qual o impacto na complexidade prática do procedimento de busca?
4. Crie um método `toArray` na classe `LinkedList` que retorne um vetor com os elementos da lista encadeada. Implemente a operação inversa na classe `ArrayList`.

5. Como seria uma lista posicional implementada baseada em vetores (`ArrayPos`
   Manter apenas o índice onde o elemento está armazenado é suficiente para mapear sua posição? Implemente essa estrutura de dados.

6. Leia a respeito do uso de iteradores para percorrer uma lista e acessar seus elementos [Goodrich et al., 2014].

7. Resolva os seguintes exercícios de Goodrich et al. [2014]:

   R-7.1: Projete uma representação de uma lista `L` inicialmente vazia após realizar as seguintes operações: `add(0, 4)`, `add(0, 3)`, `add(0, 2)`, `add(2, 1)`, `add(1, 5)`, `add(1, 6)`, `add(3, 7)`, `add(0, 8)`.

   R-7.2: Implemente uma pilha usando um `ArrayList` para armazenamento.

   R-7.3: Implemente um deque usando um `ArrayList` para armazenamento.

   R-7.5: O `java.util.ArrayList` possui um método `trimToSize` que substitui o vetor correspondente por um com capacidade equivalente ao número de elementos atuais da lista. Implemente tal método para a verão dinâmica da classe `ArrayList`.

   R-7.7: Considere uma implementação de um `ArrayList` usando um vetor dinâmico, mas ao invés de copiar os elementos para um vetor com o dobro do tamanho (isto é, de N para $2N$) quando sua capacidade é atingida, copiamos os elementos para um vetor com $\lceil N/4 \rceil$ células adicionais, indo da capacidade N para $N + \lceil N/4 \rceil$. Mostre que ao realizar uma sequência de n operações `push` (isto é, inserindo no fim) ainda opera em tempo $O(n)$.

   R-7.8: Supondo que estamos mantendo uma coleção `C` de elementos de tal modo que, cada vez que adicionamos um novo elemento na coleção, copiamos o conteúdo de `C` em um novo `ArrayList` do tamanho exato ao necessário. Qual é o tempo de processamento de adição de n elementos em uma coleção `C` inicialmente vazia?

R-7.9: O método `add` para um vetor dinâmico tem a seguinte ineficiência: no caso em que um redimensionamento ocorre, a operação correspondente leva tempo para copiar todos os elementos do antigo vetor para o novo, e então o laço subsequente muda alguns deles para dar espaço para o novo elemento. Modifique o método `add` para, no caso de redimensionamento, os elementos copiados ficarem na sua posição final do novo vetor (ou seja, nenhuma realocação deve ser feita).

R-7.10: Reimplemente a classe `ArrayStack` usando vetores dinâmicos para suportar uma capacidade ilimitada.

R-7.12: Supondo que queremos estender nossa `PositionalList` com um método `indexOf(p)`, que retorna o índice atual do elemento armazenado na posição `p`. Mostre como implementar esse método usando apenas outros métodos da interface `PositionalList` (sem detalhes da nossa implementação `LinkedPositionalList`).

R-7.13: Supondo que queremos estender nossa `PositionalList` com um método `findPosition(e)`, que retorna a primeira posição contendo um elemento igual a `e` (ou `null` se tal posição não existir). Mostre como implementar esse método usando apenas métodos existente da interface `PositionalList` (sem detalhes da nossa implementação `LinkedPositionalList`).

R-7.14: A implementação `LinkedPositionalList` não realiza nenhuma verificação de erro para testar se uma posição `p` dada é um membro relevante da lista. Explique detalhadamente o efeito da chamada `L.addAfter(p, e)` de uma lista `L` com uma posição `p` que pertence a alguma outra lista `M`.

R-7.15: Para melhor modelar uma fila (FIFO) cujas entradas possam ser deletadas antes de chegar à frente da mesma, projete uma classe `LinkedPositionalQueue` que suporte o tipo abstrato fila, com o método `enqueue` retornando uma instância de posição e suporte

um novo método `remove(p)`, que remove e retorna o elemento associado com a posição `p` da fila. Você pode usar uma `LinkedPositionalList` para armazenamento.

R-7.18: A interface `java.util.Collection` inclui um método `contains(o)`, que retorna `true` se a coleção possui um objeto que é igual a `Object o`. Implemente tal método na classe `ArrayList`.

R-7.19: A interface `java.util.Collection` inclui um método `clear()`, que remove todos os elementos de uma coleção. Implemente tal método na classe `ArrayList`.

C-7.25: Implemente uma lista baseada em vetores com capacidade fixa e tratamento circular, que chegue a um tempo $O(1)$ para inserções e remoções no índice 0, bem como para inserções e remoções no final da lista. Sua implementação também deverá fornecer um método `get` de tempo constante.

C-7.26: Complete o exercício anterior, mas usando um vetor dinâmico com capacidade ilimitada.

C-7.35: Reimplemente a classe `ArrayQueue` usando um vetor dinâmico para suportar capacidade ilimitada. Seja cuidadoso no tratamento circular do vetor no momento do redimensionamento.

P-7.58: Desenvolva um experimento para testar a eficiência de n chamadas sucessivas ao método `add` de um `ArrayList` para vários n diferentes, sob cada um dos seguintes cenários:

a. Cada `add` acontece no índice 0.

b. Cada `add` acontece no índice `size()/2`.

c. Cada `add` acontece no índice `size()`.

d. Analise seus resultados empíricos.

P-7.60: Implemente uma classe `CardHand` que suporta uma pessoa ordenando um grupo de cartas na sua mão. O simulador deverá representar uma sequência de cartas utilizando uma lista posicional única, de tal modo que cartas do mesmo naipe são mantidas juntas. Implemente esta estrategia considerando quatro "dedos", cada um armazenando cartas de um dos naipes (copas, paus, espadas e ouros). Assim, adicionar uma nova carta na mão da pessoa ou jogar uma carta da mão pode ser feito em tempo constante. A classe deve suportar os seguintes métodos:

- `addCard(r,s)`: adiciona uma nova carta com número `r` e naipe `s` para a mão.
- `play(s)`: remove e retorna uma carta de naipe `s` da mão do jogador. Se não existir carta do naipe `s`, então remove e retorne uma carta arbitrária da mão.

Referências

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons, 6th edition.

Lafore, R. and Machado, E. V. (2004). *Estruturas de dados & Algoritmos em Java*. Ciência Moderna.

Pereira, S. d. L. (2008). Estruturas de dados fundamentais: Conceitos e aplicações.