Funcional V: estruturas de dados puramente funcionais

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

10 de julho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

CC BY SA

Crédito de parte das imagens, a menos se especificado: Wikipedia

Objetivos

- Aprender a projetar estruturas funcionais básicas
- Praticar com exercícios que estendem estruturas funcionais

Pilhas

Pilhas

- Listas em Lisp são representações naturais de pilhas
- Por convenção, a pilha vazia é uma lista vazia

- Empilhamento

```
1 (define push
2   (lambda (e l)
3     (cons e l)))
```

- Desempilhamento

```
1 (define pop
2   (lambda (l)
3     (if (null? l)
4         l
5         (cdr l))))
```

- Consulta

```
1 (define peek
2   (lambda (l)
3     (if (null? l)
4         l
5         (car l))))
```

```
> (push 3 (push 2 (push 1 '())))
'(3 2 1)
> (push 4 (push 3 (push 2 (push 1 null))))
'(4 3 2 1)
> (peek (push 3 (push 2 (push 1 null))))
3
> (pop (pop (push 3 (push 2 (push 1 null)))))
'(1)
```

Filas

Filas

Implementação ingênua

- **Desenfileiramento**: remover da cabeça da lista

```
1 (define dequeue
2   (lambda (l)
3     (if (null? l)
4         l
5         (cdr l))))
```

- **Consulta**

```
1 (define front
2   (lambda (l)
3     (if (null? l)
4         l
5         (car l))))
```

- **Enfileiramento**: adicionar como último elemento da lista

```
1 (define enqueue
2   (lambda (e l)
3     (append l (list e))))
```

- Desempenho?

```
> (define q (enqueue 'c (enqueue 'b (enqueue 'a null))))
> q
'(a b c)
> (front q)
'a
> (dequeue q)
'(b c)
> (enqueue 'f (enqueue 'e q))
'(a b c e f)
```

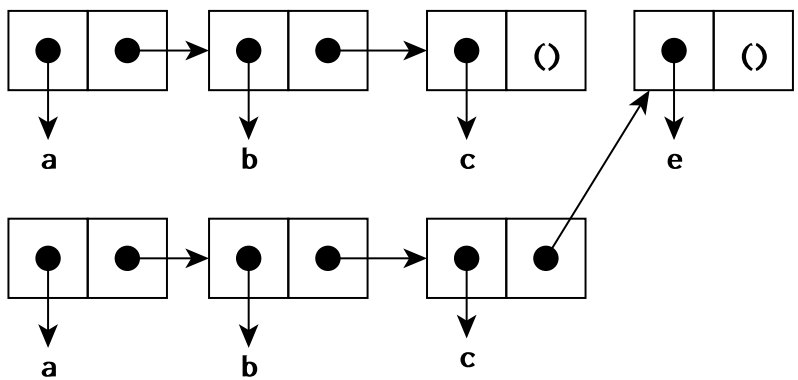
Fila: implementação ingênua

Desempenho

- O "ponto fraco" é a função `append`. Exemplo:

```
1 (define list-append
2   (lambda (a b)
3     (if (null? a)
4         b
5         (cons (car a) (list-append (cdr a) b)))))
```

- Lista da esquerda é copiada
- Lista da direita é reusada
- Overhead persistência: tamanho da lista da esquerda
- Após executar `(append '(a b c) '(e))`:



- Custo de cada enfileiramento: tamanho prévio da fila ($O(n)$)

Fila: implementação com duas pilhas

Princípios gerais

- Criar uma estrutura com duas pilhas
- Pilha A: itens em ordem normal, usada para desenfileirar
- Pilha B: itens em ordem reversa, usada para enfileirar
- Ajuste: quando a pilha A esvaziar, inverter a pilha B e substituir em A

make-queue	`(() ())`	$O(1)$
enqueue 1	`(() (1))`	$O(1)$
enqueue 2	`(() (2 1))`	$O(1)$
enqueue 3	`(() (3 2 1))`	$O(1)$
front	`((1 2 3) ())`	$O(n)$
dequeue	`((2 3) ())`	$O(1)$

- Custo *amortizado* para enfileirar um elemento é $O(3)$ (constante):
  - $O(1)$ para empilhar em B
  - $O(1)$ para desempilhar de B
  - $O(1)$ para empilhar em A

Filas

Implementação com duas pilhas

- Construtor

```
1 (define (queue-make) (cons null null))
```

- Enfileiramento

```
1 (define queue-enq
2   (lambda (e q)
3     (cons (car q) (cons e (cdr q)))))
```

- Desenfileiramento

```
1 (define queue-deq
2   (lambda (q)
3     (cond ((and (null? (car q)) (null? (cdr q))) q)
4           ((null? (car q)) (cons (cdr (reverse (cdr q))) null))
5           (else (cons (cdr (car q)) (cdr q))))))
```

Filas

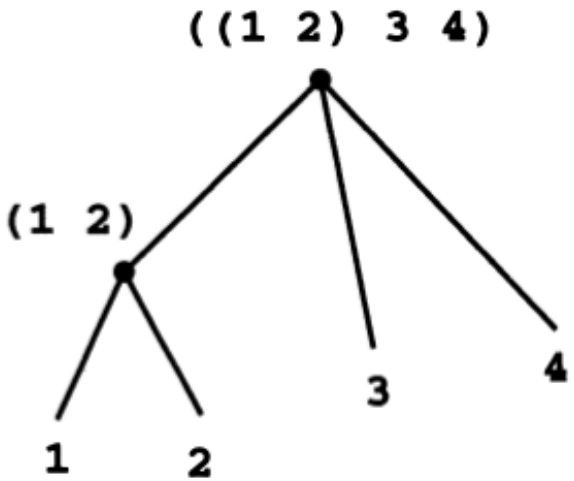
Implementação com duas pilhas

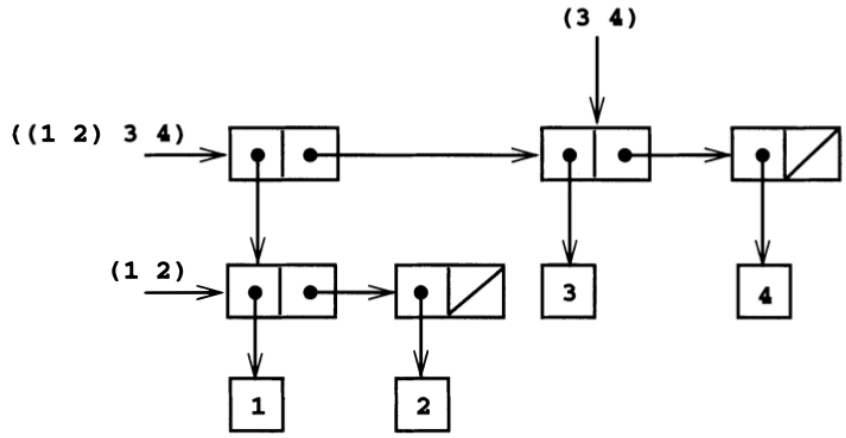
```
(define q (queue-enq 4 (queue-enq 3 (queue-enq 2 (queue-make)))))
> (() 4 3 2)
(queue-deq q)
> ((3 4))
(queue-enq 5 (queue-deq q))
> ((3 4) 5)
> (queue-enq 6 (queue-enq 5 (queue-deq q)))
'((3 4) 6 5)
```

Árvores

Árvores n-árias

- Árvores com dados apenas nas folhas (e.g. árvores de Huffman)
- Proposta de representação:
  - Um nó interno é um par
  - Um nó folha não é um par
- Exemplo: `((1 2) 3 4)`





  - Vantagem: mais simples de processar, pois há dois "tipos" de nós
  - Desvantagem: nós internos não contém dados

Árvores n-árias

Exemplo 1: contar as folhas de uma árvore

```
1 (define count-leaves
2   (lambda (t)
3     (cond ((null? t) 0)
4           ((pair? (car t)) (+ (count-leaves (car t))
5                               (count-leaves (cdr t))))
6           (else (+ 1 (count-leaves (cdr t)))))))
```

```
(count-leaves '(a b (c (d e)) f (g h)))
> 8
```

- **Exercício 1**: Defina um procedimento que multiplique todas as folhas de uma árvore por um valor.

Árvores binárias de busca

Definição e construção

- Definição:
  1. A árvore vazia é uma lista vazia
  2. Um nó é composto por valor, filho esquerdo e filho direito
  3. Filho esquerdo e filho direito são árvores
- Construtor:

```
1 (define make-node (lambda (v l r)
2                     (list v l r)))
```

- Acessores:

```
1 (define node-value car)
2 (define node-left cadr)
3 (define node-right caddr)
```

Árvores binárias de busca

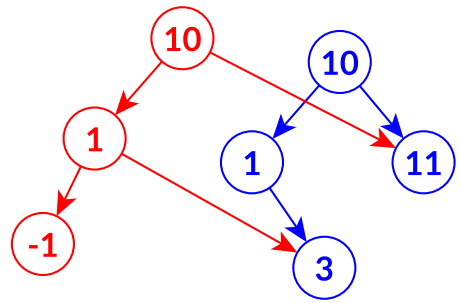
Inserção persistente

- Algoritmo de inserção recursivo (clássico):
  - **Caso 1**: árvore vazia; criar novo nó
  - **Caso 2**: valor maior que o nó atual; inserir rec. no filho direito
  - **Caso 3**: valor menor que o nó atual; inserir rec. no filho esquerdo
- Se fosse imperativo: novo nó como filho do nó-folha (atualiza referências/ponteiros)
- Como fazer declarativamente?
- **Persistência**: recriar todo o caminho percorrido até o nó-folha

Árvores binárias de busca

Exemplo: inserção persistente

```
(define t (tree-insert
            3
            (tree-insert
              11
              (tree-insert
                1
                (make-node 10 '() '())))))
t
> (10 (1 () (3 () ())) (11 () ()))
(tree-insert -1 t)
> (10 (1 (-1 () ()) (3 () ())) (11 () ()))
```



- Persistência (versionamento incremental):
  - O caminho da inserção é duplicado
  - Subárvores não-visitadas são reusadas
- **Exercício 2**: Considerando o overhead gerado pela persistência da inserção numa ABB, qual o pior caso? E o caso médio? Analisar em espaço e em tempo.

Árvore binária de busca

Algoritmo de inserção

```
1  (define tree-insert
2    (lambda (n t)
3      (cond ((null? t) (make-node n '() '()))
4            ((> n (node-value t))
5             (make-node (node-value t)
6                        (node-left t)
7                        (tree-insert n (node-right t))))
8            (else
9              (make-node (node-value t)
10                        (tree-insert n (node-left t))
11                        (node-right t))))))
```

Exercícios

Exercícios

Em todos os exercícios, não é permitido usar procedimentos intrínsecos da Racket que resolvam diretamente o problema enunciado.

3. Defina o procedimento `(queue-front q)` que retorna o elemento da frente de uma fila implementada com duas pilhas.
4. Defina um procedimento que receba uma árvore n-ária, com dados nas folhas, e que transforme a árvore em uma lista simples. Por exemplo, `'((1 2 (3 4)) (5) 6)` torna-se `'(1 2 3 4 5 6)`.
5. Um conjunto pode ser representado como uma lista de elementos sem repetições, i.e. `(1 2 3)`. Podemos enumerar os subconjuntos de um conjunto também como uma lista, i.e. `'(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))`. Construa um procedimento que gere os subconjuntos de um conjunto.
6. Defina um predicado que verifique se uma ABB contém um valor passado como argumento.
7. Construa um procedimento que gere uma lista correspondente ao percurso in-ordem de uma árvore binária de busca. Define procedimentos análogos para pré-ordem e pós-ordem.

Recursos recomendados

- Seções 2.2 e 2.3: Abelson, Harold, and Gerald Jay Sussman. “Structure and interpretation of computer programs” 2nd edition (1996). URL: `https://mitpress.mit.edu/sicp/full-text/book/book.html`
- Burch, Carl. “Persistent data structures” (2012). URL: `http://www.toves.org/books/persist/`
- Para quem tem curiosidade sobre estruturas de dados funcionais, a referência clássica é:
  - Okasaki, Chris. “Purely functional data structures” (1999). 232p. Cambridge University Press.

Funcional V: estruturas de dados puramente funcionais

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

10 de julho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição

CC BY SA

Crédito de parte das imagens, a menos se especificado: Wikipedia