Estrutura de Dados Cheat Sheet

Daniel Saad Nogueira Nunes

Construtores

Os construtores são funções que indicam à estrutura de dados genérica, como construir o tipo requisitado. Eles são passados na inicialização das estruturas de dados e devem possuir esta assinatura:

```
void* construtor(void* data);
```

Uma implementação de construtor geralmente recebe este dado qualquer, aloca espaço para uma cópia deste dado, copia as informações para a área de memória recém alocada, e retorna esta área alocada através de um ponteiro `void*`.

Destrutores

Os destrutores fazem o inverso do construtor. Eles recebem esta área que foi alocada e realiza os procedimentos para liberar o espaço utilizado. Possuem a seguinte assinatura:

```
void destrutor(void* data);
```

Comparadores

Comparadores são utilizados em EDs genéricas para especificar como comparar dois elementos. Eles possuem a seguinte assinatura:

```
int comparator(const void* a, const void* b);
```

Utiliza-se a seguinte semântica para os comparadores:

- Retornam -1, se `a` $<$ `b`
- Retornam 0, se `a` $=$ `b`
- Retornam 1, se `a` $>$ `b`

Os comparadores são utilizados em muitas estruturas que dependem de uma ordem, como as filas de prioridade e as árvores binárias de pesquisa.

Lista encadeada (list)

```
/* Tipo nó lista */
typedef struct list_node_t{
    void* data;
    struct list_node_t* next;
}list_node_t;

/* Tipo lista */
typedef struct list_t{
    list_node_t* head;
    list_node_t* tail;
    list_node_constructor_fn constructor;
    list_node_destructor_fn destructor;
    size_t size;
}list_t;

/* Inicializa uma lista ligada. */
void list_initialize(list_t** l,
    list_node_constructor_fn constructor,
    list_node_destructor_fn destructor);
/* Deleta a lista e todos os seus elementos. */
void list_delete(list_t** l);
/* Insere um elemento na lista em uma
 * posição especificada. */
void list_insert(list_t* l,void* data,size_t i);
/* Insere um elemento na cabeça da lista. */
void list_prepend(list_t* l,void* data);
/* Insere um elemento na cauda da lista. */
void list_append(list_t* l, void* data);
/* Remove um elemento da lista
 * em uma posição específica. */
void list_remove(list_t* l,size_t i);
/* Remove a cabeça da lista. */
void list_remove_head(list_t* l);
/* Remove a cauda da lista. */
void list_remove_tail(list_t* l);
/* Acessa o elemento de uma
 * posição específica da lista. */
void* list_access(list_t* l,size_t i);
/* Acessa a cabeça da lista. */
void* list_access_head(list_t* l);
/* Acessa a cauda da lista */
void* list_access_tail(list_t* l);
/* Indica o tamanho da lista. */
size_t list_size(list_t* l);
/* Verifica se a lista está vazia */
size_t list_empty(list_t* l);
```

Lista duplamente encadeada (dlist)

```
/* Tipo nó de lista duplamente encadeada */
typedef struct dlist_node_t{
    void* data;
    struct dlist_node_t* next;
    struct dlist_node_t* prev;
}dlist_node_t;

/* Tipo lista duplamente encadeada */
typedef struct dlist_t{
    dlist_node_t* head;
    dlist_node_t* tail;
    dlist_node_constructor_fn constructor;
    dlist_node_destructor_fn destructor;
    size_t size;
}dlist_t;

/* Inicializa a dlista */
void dlist_initialize(dlist_t** l,
    dlist_node_constructor_fn constructor,
    dlist_node_destructor_fn destructor);
/* Deleta a dlista */
void dlist_delete(dlist_t** l);
/* Insere um elemento na dlista na posição i */
void dlist_insert(dlist_t* l,void* data,size_t i);
/* Anexa um elemento na cabeça da dlista */
void dlist_prepend(dlist_t* l,void* data);
/* Anexa um elemento na cauda da dlista */
void dlist_append(dlist_t* l, void* data);
/* Remove o elemento da posição i da dlista */
void dlist_remove(dlist_t* l,size_t i);
/* Remove a cabeça da dlista */
void dlist_remove_head(dlist_t* l);
/* Remove a cauda da dlista */
void dlist_remove_tail(dlist_t* l);
/* Acessa o conteúdo da dlista na posição i */
void* dlist_access(dlist_t* l,size_t i);
/* Acessa o conteúdo da cabeça */
void* dlist_access_head(dlist_t* l);
/* Acessa o conteúdo da cauda */
void* dlist_access_tail(dlist_t* l);
/* Retorna o tamanho da dlista */
size_t dlist_size(dlist_t* l);
/* Retorna se a dlista está vazia ou não */
size_t dlist_empty(dlist_t* l);
```

Fila (queue)

```
/* Tipo fila */
typedef struct queue_node_t{
    void* data;
    struct queue_node_t* next;
}queue_node_t;

/* Tipo nó fila */
typedef struct queue_t{
    queue_node_t* front;
    queue_node_t* back;
    queue_node_constructor_fn constructor;
    queue_node_destructor_fn destructor;
    size_t size;
}queue_t;

/* Inicializa fila */
void queue_initialize(queue_t** q,
    queue_node_constructor_fn constructor,
    queue_node_destructor_fn destructor);
/* Destroi fila */
void queue_delete(queue_t** q);
/* Retira o elemento da frente da fila */
void queue_pop(queue_t* q);
/* Insere o elemento na traseira da fila */
void queue_push(queue_t* q,void* data);
/* Retorna a frente da fila */
void* queue_front(queue_t* q);
/* Retorna o tamanho da fila */
size_t queue_size(queue_t* q);
/* Retorna se a fila está vazia ou não */
size_t queue_empty(queue_t* q);
```

Deque (deque)

```
/* Tipo nó deque */
typedef struct deque_node_t{
    struct deque_node_t* next;
    struct deque_node_t* prev;
    void* data;
}deque_node_t;

/* Tipo deque */
typedef struct deque_t{
    struct deque_node_t* front;
    struct deque_node_t* back;
    deque_node_constructor_fn constructor;
    deque_node_destructor_fn destructor;
    size_t size;
}deque_t;

/* Inicializa a estrutura deque. */
void deque_initialize(deque_t** d,
    deque_node_constructor_fn constructor,
    deque_node_destructor_fn destructor);
/* Deleta a estrutura de dados e  libera toda memória
 * alocada desta estrutura. */
void deque_delete(deque_t** d);
/* Insere um elemento no início do deque. */
void deque_push_front(deque_t* d,void* data);
/* Insere um elemento no final do deque */
void deque_push_back(deque_t* d,void* data);
/* Remove um elemento do início do deque. */
void deque_pop_front(deque_t* d);
/* Remove um elemento do final do deque. */
void deque_pop_back(deque_t* d);
/* Acessa o elemento do início do deque. */
void* deque_front(deque_t* d);
/* Acessa o elemento do final do deque. */
void* deque_back(deque_t* d);
/* Indica o tamanho do deque. */
size_t deque_size(deque_t* d);
/* Verifica se o deque está vazio. */
size_t deque_empty(deque_t* d);
```

Fila de prioridades (priority_queue)

```
/* Tipo fila de prioridades */
typedef struct priority_queue_t{
    void** data;
    priority_queue_element_constructor_fn constructor;
    priority_queue_element_destructor_fn destructor;
    priority_queue_element_compare_fn comparator;
    size_t size;
    size_t capacity;
}priority_queue_t;

/* Inicializa a estrutura priority_queue  */
void priority_queue_initialize(priority_queue_t** pq,
    priority_queue_element_constructor_fn constructor,
    priority_queue_element_destructor_fn destructor,
    priority_queue_element_compare_fn comparator);
/* Deleta a estrutura de dados e libera toda memória
 * alocada desta estrutura. */
void priority_queue_delete(priority_queue_t** pq);
/* Insere um elemento na fila */
void priority_queue_push(priority_queue_t* pq,void* data);
/* Acessa o elemento do início da priority_queue */
void* priority_queue_front(priority_queue_t* pq);
/* Remove um elemento do início da priority_queue. */
void priority_queue_pop(priority_queue_t* pq);
/* Indica o tamanho da priority_queue. */
size_t priority_queue_size(priority_queue_t* pq);
/* Verifica se a priority_queue está vazia. */
size_t priority_queue_empty(priority_queue_t* pq);
```

Pilha (stack)

```
/* Tipo nó pilha */
typedef struct stack_node_t{
    void* data;
    struct stack_node_t* next;
}stack_node_t;

/* Tipo pilha */
typedef struct stack_t{
    stack_node_t* top; /*Topo da pilha*/
    stack_node_constructor_fn constructor;
    stack_node_destructor_fn destructor;
    size_t size; /*tamanho da pilha*/
}stack_t;

/* Inicializa pilha */
void stack_initialize(stack_t** s,
    stack_node_constructor_fn constructor,
    stack_node_destructor_fn destructor);
/* Destroi pilha */
void stack_delete(stack_t** s);
/* Retira o elemento do topo da pilha */
void stack_pop(stack_t* s);
/* Insere o elemento no topo da pilha */
void stack_push(stack_t* s,void* data);
/* Retorna o topo da pilha */
void* stack_top(stack_t* s);
/* Retorna o tamanho da pilha */
size_t stack_size(stack_t* s);
/* Retorna se a pilha está vazia ou não */
size_t stack_empty(stack_t* s);
```

BST (bst)

```
/* Tipo nó BST */
typedef struct bst_node_t{
    void* data;
    struct bst_node_t* left;
    struct bst_node_t* right;
}bst_node_t;

/* Tipo BST */
typedef struct bst_t{
    bst_node_t* root;
    bst_element_constructor_fn constructor;
    bst_element_destructor_fn destructor;
    bst_tree_element_compare_fn comparator;
    size_t size;
}bst_t;

/* Inicializa a BST */
void bst_initialize(bst_t** t,
    bst_element_constructor_fn constructor,
    bst_element_destructor_fn destructor,
    bst_tree_element_compare_fn comparator);
/* Deleta a estrutura de dados e libera toda memória
 * alocada desta estrutura. */
void bst_delete(bst_t**);
/* Insere um elemento na árvore */
void bst_insert(bst_t* t,void* data);
/* Remove o elemento da árvore que ao compara-lo
 * com o valor data o comparador retorne 0 */
void bst_remove(bst_t* t,void* data);
/* Retorna 1 se existe um elemento na arvore
que corresponde ao valor de data e 0 caso não */
int bst_find(bst_t*, void* data);
/* Indica o tamanho da árvore */
size_t bst_size(bst_t* t);
```

Treap (treap)

```
/* Inicializa a Treap */
/* Tipo nó treap */
typedef struct treap_node_t{
    size_t priority;
    void* data;
    struct treap_node_t* left;
    struct treap_node_t* right;
}treap_node_t;

/* Tipo árvore treap */
typedef struct treap_t{
    treap_node_t* root;
    size_t size;
    treap_element_constructor_fn constructor;
    treap_element_destructor_fn destructor;
    treap_tree_element_compare_fn comparator;
}treap_t;

void treap_initialize(treap_t** t,
    treap_element_constructor_fn constructor,
    treap_element_destructor_fn destructor,
    treap_tree_element_compare_fn comparator);
/* Deleta a estrutura de dados e libera toda memória
 * alocada desta estrutura. */
void treap_delete(treap_t**);
/* Insere um elemento na árvore */
void treap_insert(treap_t*,void* data);
/* Remove o elemento da árvore que ao compara-lo
 * com o valor data o comparador retorne 0 */
```

```
void treap_remove(treap_t*,void* data);
/* Retorna 1 se existe um elemento na arvore
que corresponde ao valor de data e 0 caso não */
int treap_find(treap_t*,void* data);
/* Indica o tamanho da árvore */
size_t treap_size(treap_t*);
```

Árvore AVL (avl_tree)

```
/* Tipo nó AVL */
typedef struct avl_node_t{
    void* data;
    size_t height;
    struct avl_node_t* left;
    struct avl_node_t* right;
}avl_node_t;
```

```
/* Tipo árvore AVL */
typedef struct avl_tree_t{
    struct avl_node_t* root;
    avl_tree_element_constructor_fn constructor;
    avl_tree_element_destructor_fn destructor;
    avl_tree_element_compare_fn comparator;
    size_t size;
}avl_tree_t;

/* Inicializa a AVL Tree */
void avl_tree_initialize(avl_tree_t** t,
    avl_tree_element_constructor_fn constructor,
    avl_tree_element_destructor_fn destructor,
    avl_tree_element_compare_fn comparator);
```

```
/* Deleta a estrutura de dados e libera toda memória
 * alocada desta estrutura. */
void avl_tree_delete(avl_tree_t** t);
/* Insere um elemento na árvore */
void avl_tree_insert(avl_tree_t* t,void* data);
/* Remove o elemento da árvore que ao compara-lo
 * com o valor data o comparador retorne 0 */
void avl_tree_remove(avl_tree_t* t,void* data);
/* Retorna 1 se existe um elemento na arvore
que corresponde ao valor de data e 0 caso não */
int avl_tree_find(avl_tree_t* t, void* data);
/* Indica o tamanho da árvore */
size_t avl_tree_size(avl_tree_t* t);
```

Inspirado em http://wch.github.io/latexsheet/ by Winston Chang