Anotações referente a Aula 1 de Estruturas de Dados 2 com Bruno Ribas

Conceito Geral

Uma **Symbol Table (ST)** é uma estrutura de dados usada para armazenar pares **chave-valor** (ou itens). Cada item tem:

- Uma **chave (key)**: usada para identificar o item.
- Um **dado (data)**: informação associada à chave.

Nas suas anotações, parece que você está explorando:

1. **Funções básicas** de uma ST (como init, insert, remove, etc.).
2. **Definições de tipos** (como item, key).
3. **Implementações**: usando array com acesso direto e lista encadeada.

Classificação:

ST init

- **Descrição**: Inicializa a estrutura (ex.: aloca memória para o array ou define o início da lista).
- **Classificação**: **Função**
- **Motivo**: Nas suas anotações, STinit() retorna 1 (sucesso) ou 0 (falha), indicando um resultado.

ST insert

- **Descrição**: Insere um item na estrutura.
- **Classificação**: **Procedimento**
- **Motivo**: Não retorna um valor explícito; apenas modifica a estrutura adicionando o item (ex.: st[st_last++] = ni).

ST remove

- **Descrição**: Remove um item com uma chave específica.
- **Classificação**: **Procedimento**
- **Motivo**: Apenas altera a estrutura (ex.: troca com o último item e decrementa st_last), sem retorno explícito.

ST search

- **Descrição**: Busca um item pela chave.
- **Classificação**: **Função**

- **Motivo**: Retorna algo (ex.: índice no array ou o item em acesso direto), como visto em return st[i] ou return i.

ST empty

- **Descrição**: Verifica se a estrutura está vazia.
- **Classificação**: **Função**
- **Motivo**: Retorna um valor booleano (1 para vazio, 0 para não vazio), como em return st_last == 0.

ST count

- **Descrição**: Retorna o número de itens na estrutura.
- **Classificação**: **Função**
- **Motivo**: Retorna um valor numérico (ex.: return st_last).

ST destroy

- **Descrição**: (Não detalhado nas anotações) Provavelmente libera a memória da estrutura.
- **Classificação**: **Procedimento**
- **Motivo**: Normalmente apenas desaloca memória (ex.: free(st)), sem retorno.

ST copy

- **Descrição**: (Não detalhado nas anotações) Provavelmente copia a estrutura para outra.
- **Classificação**: **Procedimento**
- **Motivo**: Típico de apenas criar uma cópia, sem retorno explícito (a menos que retorne a nova estrutura, mas não há indício disso).

ST duplicate

- **Descrição**: (Não detalhado nas anotações) Talvez duplique itens ou a estrutura.
- **Classificação**: **Procedimento**
- **Motivo**: Similar a ST copy, geralmente apenas modifica ou cria algo sem retorno.

ST sort

- **Descrição**: (Não detalhado nas anotações) Ordena os itens na estrutura.
- **Classificação**: **Procedimento**
- **Motivo**: Apenas reorganiza os itens na estrutura, sem retorno explícito.

ST clear

- **Descrição**: (Não detalhado nas anotações) Provavelmente limpa todos os itens.
- **Classificação**: **Procedimento**
- **Motivo**: Apenas reseta a estrutura (ex.: zera st_last ou redefine como vazia), sem retorno.

Exemplos:

```
#include <stdio.h>
#include <stdlib.h>

// Definição do tipo Item
typedef struct {
    int k;    // Chave (key)
    char d;   // Dado (data), usando char para simplicidade
} Item;

// Macros das anotações
#define key(A) (A.k)
#define less(A, B) (key(A) < key(B))
#define eq(A, B) (key(A) == key(B))
#define stswap(A, B) { Item tmp = A; A = B; B = tmp; }
#define maxItems 10  // Tamanho pequeno para exemplo

// Variáveis globais
Item *st;
int st_last = 0;
```

ST init:

```
int STinit() {
    st = malloc(sizeof(Item) * maxItems);
    if (st == NULL) {
        printf("Erro: falha na alocação de memória!\n");
        return 0;
    }
    st_last = 0;
    printf("ST inicializada com sucesso!\n");
    return 1;
}
```

Propósito: Inicializa a ST alocando memória para o array.
st = malloc(sizeof(Item) * maxItems);: Aloca memória para maxItems (10) elementos do tipo Item. Cada Item tem tamanho sizeof(Item) (geralmente 8 bytes: 4 para int k + 1 para char d + padding).
if (st == NULL): Verifica se a alocação falhou (retorna NULL se não houver memória disponível).
printf("Erro: falha na alocação de memória!\n"); return 0;: Exibe erro e retorna 0 (falha).
st_last = 0;: Zera o contador de itens, indicando que a ST começa vazia.

printf("ST inicializada com sucesso!\n"); return 1;: Confirma sucesso e retorna 1.

STInsert:

```
void STinsert(Item ni) {
   if (st_last < maxItems) {
      st[st_last++] = ni;
      printf("Inserido: chave %d, dado %c\n", ni.k, ni.d);
   } else {
      printf("Erro: ST cheia!\n");
   }
}
```

Propósito: Insere um novo item na próxima posição livre.
if (st_last < maxItems): Verifica se há espaço no array (limite é maxItems).
st[st_last++] = ni;: Copia o item ni para a posição st_last e incrementa st_last (pós-incremento: usa o valor atual e depois soma 1).
printf("Inserido: chave %d, dado %c\n", ni.k, ni.d);: Confirma a inserção com os valores.
else { printf("Erro: ST cheia!\n"); }: Se st_last atingir maxItems, exibe erro (não insere).

STRemove:

```
void STremove(int r) {
   int i;
   for (i = 0; i < st_last; i++) {
      if (eq(r, st[i])) {
         stswap(st[i], st[st_last - 1]);
         st_last--;
         printf("Removido: chave %d\n", r);
         return;
      }
   }
   printf("Erro: chave %d não encontrada!\n", r);
}
```

STSearch:

Propósito: Busca um item pela chave e retorna seu índice.
Código:

```
int STsearch(int s) {
   for (int i = 0; i < st_last; i++) {
```

```
        if (eq(s, st[i])) {
            printf("Encontrado: chave %d, dado %c\n", st[i].k, st[i].d);
            return i;
        }
    }
    printf("Chave %d não encontrada!\n", s);
    return -1;
}
```

STEmpty:

Propósito: Verifica se a ST está vazia.
Código:

```
int STempty() {
    return st_last == 0;
}

int main() {
    STinit();
    printf("Vazia? %s\n", STempty() ? "Sim" : "Não");  // "Sim"
    STinsert((Item){3, 'A'});
    printf("Vazia? %s\n", STempty() ? "Sim" : "Não");  // "Não"
    return 0;
}
```

ST count

- **Propósito**: Retorna o número de itens na ST.
- **Código**:

```
int STcount() {
    return st_last;
}

int main() {
    STinit();
    printf("Itens: %d\n", STcount());  // "Itens: 0"
    STinsert((Item){3, 'A'});
    STinsert((Item){1, 'B'});
    printf("Itens: %d\n", STcount());  // "Itens: 2"
    return 0;
}
```

ST destroy

- **Propósito**: Libera a memória da ST.
- **Código**:

```
void STdestroy() {
    free(st);
    st = NULL;
    st_last = 0;
    printf("ST destruída!\n");
}


int main() {
    STinit();
    STinsert((Item){3, 'A'});
    STdestroy();
    printf("Itens após destruir: %d\n", STcount());  // "Itens: 0"
    return 0;
}
```

ST copy

- **Propósito**: Copia a ST para outro array.
- **Código**:

```
Item* STcopy() {
    Item *new_st = malloc(sizeof(Item) * maxItems);
    if (new_st == NULL) {
        printf("Erro: falha ao copiar ST!\n");
        return NULL;
    }
    for (int i = 0; i < st_last; i++) {
        new_st[i] = st[i];
    }
    printf("ST copiada com sucesso!\n");
    return new_st;
}

int main() {
    STinit();
    STinsert((Item){3, 'A'});
    STinsert((Item){1, 'B'});
```

```
    Item *copia = STcopy();
    printf("Cópia: [%d, %c], [%d, %c]\n", copia[0].k, copia[0].d, copia[1].k, copia[1].d);
    free(copia);
    STdestroy();
    return 0;
}
```

ST duplicate

- **Propósito**: Duplica os itens existentes na ST.
- **Código**:

```
void STduplicate() {
    int original_size = st_last;
    if (st_last * 2 <= maxItems) {
        for (int i = 0; i < original_size; i++) {
            st[st_last++] = st[i];
        }
        printf("Itens duplicados!\n");
    } else {
        printf("Erro: espaço insuficiente para duplicar!\n");
    }
}

int main() {
    STinit();
    STinsert((Item){3, 'A'});
    STinsert((Item){1, 'B'});
    printST();  // [{3, A}, {1, B}]
    STduplicate();
    printST();  // [{3, A}, {1, B}, {3, A}, {1, B}]
    STdestroy();
    return 0;
}
```

ST sort

- **Propósito**: Ordena os itens por chave.
- **Código** (usando bubble sort simples):

```
void STduplicate() {
    int original_size = st_last;
    if (st_last * 2 <= maxItems) {
```

```
        for (int i = 0; i < original_size; i++) {
            st[st_last++] = st[i];
        }
        printf("Itens duplicados!\n");
    } else {
        printf("Erro: espaço insuficiente para duplicar!\n");
    }
}

int main() {
    STinit();
    STinsert((Item){3, 'A'});
    STinsert((Item){1, 'B'});
    printST();  // [{3, A}, {1, B}]
    STduplicate();
    printST();  // [{3, A}, {1, B}, {3, A}, {1, B}]
    STdestroy();
    return 0;
}
```

ST clear

- **Propósito**: Limpa todos os itens da ST.
- **Código**:

```
void STclear() {
    st_last = 0;
    printf("ST limpa!\n");
}


int main() {
    STinit();
    STinsert((Item){3, 'A'});
    STinsert((Item){1, 'B'});
    printST();  // [{3, A}, {1, B}]
    STclear();
    printST();  // []
    STdestroy();
    return 0;
}
```