UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
Departamento de Ciências de Computação

SCC0122 Estruturas de Dados
Prof. Thiago A. S. Pardo

Prova 1
16/10/2020

Responda as questões da prova no editor de texto de sua preferência e submeta a resolução no e-Disciplinas (aba "Avaliações"). Se desejar, você pode responder nesse próprio documento. Opcionalmente, você também pode responder em papel e submeter um arquivo com a digitalização/foto da prova.

Nome do aluno: Gustavo Siqueira Barbosa

Número USP: 10728122

1) (1,0) Marque como verdadeira (V) ou falsa (F) cada uma das afirmações abaixo.

[v] Estruturas de dados e TADs não são termos equivalentes.
[v] Quaisquer listas, lineares ou não, podem ser implementadas como TADs.
[F] O uso de TADs implica em uma maior complexidade de tempo do programa que o utiliza.
[F] Todos os programas organizados em arquivos .h e .c seguem os princípios de TAD.
[F] Tipos de dados e TADs são termos equivalentes.
[F] Toda lista é um vetor tradicional em C, usualmente declarado como int v[tamanho_qualquer].
[F] Toda lista é uma fila ou uma pilha.
[F] Toda lista é sequencial.
[v] Toda fila é uma lista.
[v] Um vetor é uma lista.

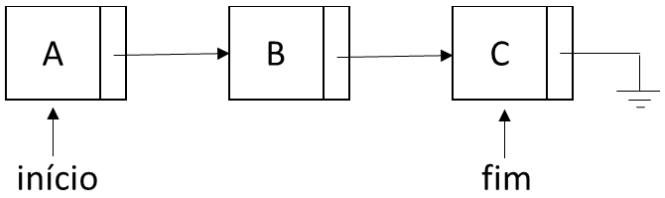
2) (1,0) Diga quais são as vantagens e desvantagens de estruturas de dados alocadas dinamicamente em relação às alocadas estaticamente. Resposta nas últimas páginas

3) Considere que você está implementando um sistema que gerencia uma pilha com informações sobre provas já corrigidas. Cada prova é identificada pelo nome do aluno, número USP e a nota tirada.

(a) (1,0) Declare em C a estrutura de dados dessa pilha, considerando que a estrutura é estática e sequencial. Faça as suposições adicionais que julgar necessárias.

(b) (2,0) Usando a estrutura de dados que declarou no item anterior, implemente em C as funções básicas de criar/inicializar a pilha, push e pop. Se desejar, pode implementar outras funções de suporte. Lembre-se das boas práticas de programação. Faça as suposições que julgar necessárias.

4) Em estruturas de dados, um "deque" (do inglês, *double-ended queue*) é uma fila de duas extremidades, em que é possível inserir e remover elementos das duas extremidades da fila. Considere, por exemplo, a estrutura dinâmica e encadeada abaixo:



Nessa estrutura, tanto A (na extremidade esquerda) quanto C (na extremidade direita) poderiam ser removidos. De forma similar, um novo elemento D poderia ser incluído tanto à esquerda de A (passando a ser o novo início da fila, portanto) quanto à direita de C (passando a ser o novo fim da lista). A escolha de onde inserir ou remover cada elemento (se à esquerda ou à direita do deque) depende da aplicação visada e da vontade do usuário que está utilizando a estrutura.

(a) (1,0) Declare em C a estrutura de dados desse deque, considerando que a estrutura é dinâmica e encadeada e armazena números inteiros. Suponha que sua estrutura é similar à que foi desenhada na questão, ou seja, cada bloco tem um campo de informação e um ponteiro para o próximo elemento. Também há ponteiros para indicar o início e o fim do deque. Faça as suposições adicionais que julgar necessárias.

(b) (1,0) Usando a estrutura de dados que declarou no item anterior, implemente em C as funções básicas entrar_na_esquerda e sair_da_esquerda do deque. Se desejar, pode implementar outras funções de suporte. Faça as suposições que julgar necessárias.

(c) (1,0) Implemente agora (também em C) as funções básicas entrar_na_direita e sair_da_direita do deque. Novamente, se desejar, pode implementar outras funções de suporte. Faça as suposições que julgar necessárias.

(d) (1,0) Implemente em C uma função iterativa que some os elementos do deque e retorne esse valor, percorrendo o deque da esquerda para a direita.

(e) (1,0) Implemente agora (também em C) uma função recursiva que some os elementos do deque e retorne esse valor, percorrendo o deque da esquerda para a direita.

2) Estruturas de dados alocadas dinamicamente nos dão mais possibilidades de manipulação. É possível alterar seu tamanho durante o tempo de execução, o que é muito vantajoso, por vezes essencial, a depender da aplicação a ser desenvolvida. Isso permite evitar gasto desnecessário de memória. Entretanto, essa maior possibiliade de manipulação trás consigo um custo de uma implementação mais complexa, que demanda maior cautela durante o desenvolvimento. Esse custo faz com que, para determinadas aplicações, seja mais vantajoso optar por se trabalhar com estruturas estáticas a dinâmicas.

3)

```
typedef struct prova_{
    char *nome_aluno;
    char *n_usp;
    float nota;
}Prova;

typedef struct pilha_{
    Prova **provas;
    int topo;
    int tamanho;
}Pilha;

//Função de criar recebe o tamanho máximo e retorna um ponteiro para a pilha
Pilha* CriarPilha(int tam){
    Pilha *P = (Pilha *) malloc(sizeof(Pilha));
    P->provas = (Prova **) calloc(tam, sizeof(Prova*));
    for (int i=0; i < tam; i++)
        provas[i] = (Prova *) calloc(1, sizeof(Prova));


    P->topo = 0;
    P->tamanho = tam;

    return P;
}

//Funções EstaCheia e EstaVazia foram implementadas para auxiliar no processo de checagem
int EstaCheia(Pilha *P){
    if (P->topo == P->tamanho) return 1;
    else return 0;
}

int EstaVazia(Pilha *P){
    if (P->topo == 0) return 1;
    else return 0;
}
```

```
//Retorna um valor inteiro referente a um indicador de erro
//Fiquei em dúvida se era para implementar a inserção dos dados das provas, então fiz esta funçã
Prova* PreencherEstrutura(){

    Prova *prova = (Prova *) malloc(sizeof(Prova));
    prova->nome = (char *) calloc(sizeof(char) * 500);
    prova->n_usp = (char *) calloc(sizeof(char) * 15);

    printf("Insira o nome: ")
    scanf(" %s",prova->nome);
    printf("Insira o numero usp: ")
    scanf(" %s",prova->n_usp);
    printf("Insira nota: ")
    scanf(" %d",prova->nota);

    return prova;
}

//Retorna um valor inteiro referente a um indicador de erro
int Push(Pilha *P, Prova *prova){

    if(EstaCheia(P)) return 1;

    P->provas[P->topo] = prova;
    P->topo++;

    return 0;
}

//Retorna um valor inteiro referente a um indicador de erro
int Pop(Pilha *P, Prova *prova){

    if(EstaVazia(P)) return 1;

    prova = P->provas[P->topo];
    P->topo--;

    return 0;
}
```

4)

```
typedef struct node_{
    elem val;
    struct node_ *next;
}Node;

typedef struct list_{
    Node *first;
    Node *last;
    int n_elem;
}List;

int IsEmpty(List *list){
    if (list->n_elem == 0) return 1;
    else return 0;
}

int entrar_na_direita(List *list, elem e){
    Node *aux_node = (Node *) malloc(sizeof(Node));
    if (aux_node == NULL) return 1;

    if (IsEmpty(list)){
        list->first = aux_node;
    }
    else {
        list->last->next = aux_node;
    }

    list->last = aux_node;

    list->last->val = e;
    list->last->next = NULL;

    list->n_elem++;

    return 0;
}

int sair_da_direita(List *list, elem *e){
    if(IsEmpty(list)) return 1;
    *e = list->last->val;
    Node *aux_node = list->first;

    while(aux_node->next != list->last)
        aux_node = aux_node->next;

    if (aux_node != NULL)
        aux_node->next = NULL;

    free(list->last);
    list->last = aux_node;
    list->n_elem--;

    return 0;
}
```

```
int entrar_na_esquerda(List *list, elem e){
    Node *aux_node = (Node *) malloc(sizeof(Node));
    if (aux_node == NULL) return 1;

    if (IsEmpty(list)){
        list->first = aux_node;
    }
    else {
        aux_node->next = list->first;
    }

    list->first = aux_node;

    list->first->val = e;

    list->n_elem++;

    return 0;
}

int sair_da_esquerda(List *list, elem *e){

    if(IsEmpty(list)) return 1;

    *e = list->first->val;

    Node *aux_node = list->first->next;

    free(list->first);
    list->first = aux_node;

    list->n_elem--;

    return 0;
}

int NumeroElemIterativo(List *list){
    Node *aux_node;
    aux_node = list->first;

    int i = 0;
    while(aux_node != NULL){
        i++;
        aux_node = aux_node->next;
    }

    return i;
}
```

```
//Para esta função funcionar corretamente, decidi fazer a soma
//separadamente
int NumeroElemRecursivo(List *list){
    int i;
    i = SomaRecursiva(list->first);
    return i;
}

int SomaRecursiva(Node *node){
    int i;
    if (node->next != NULL) i = SomaRecursiva(node->next);
    else i = 0;

    return (i+1);
}
```