Estrutura de Dados

Prof. Maurício Neto

UNIFAVIP
wyden

O que vamos ver?

01

Alocação Sequencial

02

Listas, Pilhas e filas

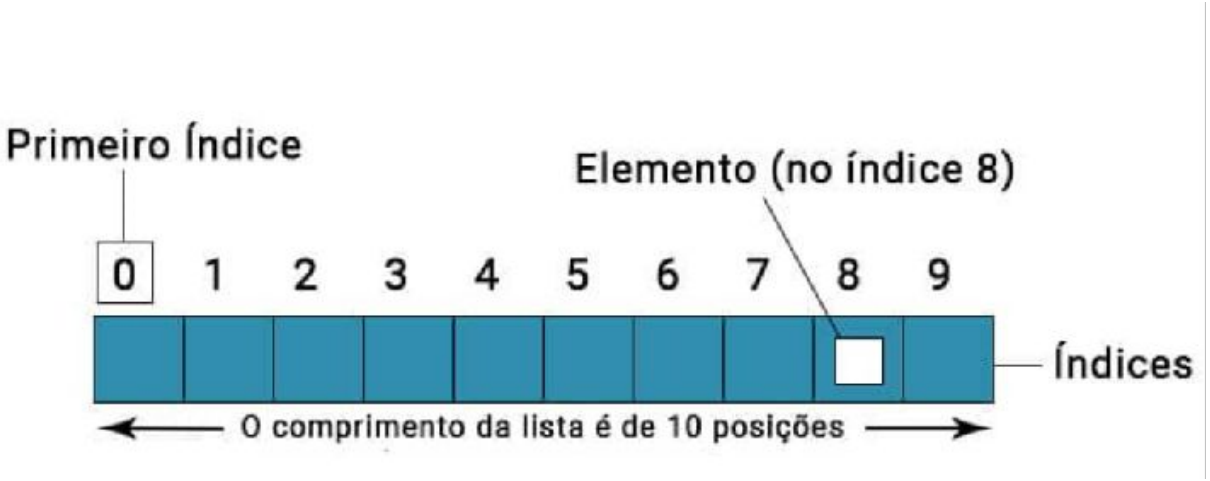
UNIFAVIP
wyden

01

Alocação sequencial

Listas, Pilhas e filas (Alocação sequencial)

É o armazenamento de dados de forma sequencial na memória do computador. Isto quer dizer que as posições de memória ocupadas serão contínuas.



UNIFAVIP wyden

Listas, Pilhas e filas

Para que a alocação sequencial possa ser levada a termo, o programa precisa informar previamente todo o tamanho de memória que será necessário.

Em linguagens de programação de alto nível, a alocação sequencial é representada pelos arrays ou vetores.

Operações em listas lineares genéricas

São estruturas de dados não primitivas, usadas para reunir um conjunto de elementos que guardam relação entre si.

Uma lista linear pode armazenar tipos de dados complexos, isto é, cada nó pode ter campos que armazenam elementos com características distintas. Pode-se designar um desses campos como sendo a chave de busca da lista, o qual é utilizado para indexar os nós e é chamado de “chave”.

As listas apresentam casos particulares, chamados de deque, pilha e fila.

Operações em listas lineares genéricas

Observe os pseudocódigos a seguir, nos quais a lista é representada por “**Lista**” e possui “**n**” posições ocupadas.

Algoritmo 1: Busca

```
1: int buscar ( chave )
2:      se n > 0
3:              para i = 1 até i <= n
4:                      se Lista [ i ].chave == chave
6:                      retornar i
7:      retornar n + 1
```

UNIFAVIP wyden

Operações em listas lineares genéricas

```c
typedef struct {
    int chave;
} Item;

int buscar(Item lista[], int n, int chave) {
    if (n > 0) {
        for (int i = 0; i < n; i++) {
            if (lista[i].chave == chave) {
                return i;
            }
        }
    }
    return n + 1;
}
```

Operações em listas lineares genéricas

Observe os pseudocódigos a seguir, nos quais a lista é representada por “**Lista**” e possui “**n**” posições ocupadas.
Algoritmo 2: Inserção

```
1: int inserir ( novo_elemento )
2:      se busca ( novo_elemento.chave ) == n + 1
3:              Lista [ n + 1 ] == novo_elemento
4:              n = n + 1
5:              retornar 1
6:      senão retornar -1
```

UNIFAVIP wyden

Operações em listas lineares genéricas

```c
int inserir(Item lista[], int *n, Item novo_elemento, int capacidade) {
    // Verifica se o novo elemento já existe na lista
    if (buscar(lista, *n, novo_elemento.chave) == (*n + 1)) {
        if (*n < capacidade) { // Verifica se há espaço para inserir
            lista[*n] = novo_elemento; // Insere o novo Elemento
            (*n)++; // Incrementa o número de elementos
            return 1; // Inserção bem-sucedida
        } else {
            return -1; // Lista cheia
        }
    } else {
        return -1; // Elemento já existe na lista
    }
}
```

Operações em listas lineares genéricas

Observe os pseudocódigos a seguir, nos quais a lista é representada por “**Lista**” e possui “**n**” posições ocupadas.

Algoritmo 3: Remoção

```
1:    int remover ( chave )
2:          se n > 0
3:                int i = busca ( chave )
4:                se i < n + 1
5:                      para a = i até a < n
6:                            Lista [ i ] = Lista [ i + 1 ]
7:                            n = n -1
8:                senão retornar -1
9:          senão retornar “Erro: lista vazia”
```

UNIFAVIP wyden

Operações em listas lineares genéricas

```c
int remover(Item lista[], int *n, int chave) {
    if (*n > 0) {
        int i = buscar(lista, *n, chave);
        if (i < *n) { // Se o elemento foi encontrado
            for (int a = i; a < *n - 1; a++) {
                lista[a] = lista[a + 1]; // Desloca os elementos à esquerda
            }
            (*n)--; // Diminui o número de elementos
            return 1; // Remoção bem-sucedida
        } else {
            return -1; // Chave não encontrada
        }
    } else {
        printf("Erro: lista vazia.\n");
        return -1; // Lista vazia
    }
}
```

Operações em listas lineares genéricas

Código 2: Alocação dinâmica

```
   [...]
1:  int *vetor;
2:  vetor = ( int * ) malloc ( tamanho_vetor * sizeof ( int ) );
   [...]
```

A função malloc solicita ao sistema operacional que reserve, em tempo de execução, uma área contígua de memória igual à “tamanho_vetor” * o tamanho do tipo inteiro. A variável “tamanho_vetor” pode ser determinada durante a execução do programa.

UNIFAVIP wyden

Operações em listas lineares genéricas

```c
// Aloca dinamicamente o vetor
vetor = (int*) malloc(tamanho_vetor * sizeof(int));

// Verifica se a alocação foi bem-sucedida
if (vetor == NULL) {
    printf("Erro ao alocar memória.\n");
    return 1;
}
```

Malloc

A função malloc (memory allocation) é usada para alocar memória dinamicamente em C. Isso significa que podemos solicitar espaço na memória em tempo de execução, em vez de especificar o tamanho do armazenamento de dados (como arrays) no momento da compilação.

```
ptr = (tipo_dado*) malloc(tamanho_em_bytes);
```

ptr: É um ponteiro que vai armazenar o endereço da memória alocada.
tipo_dado*: A conversão de tipo (cast) que define o tipo de dados que você está alocando (ex.: int*, float*, etc.).
malloc(tamanho_em_bytes): Solicita um bloco de memória de tamanho especificado em bytes.

Malloc

10 * sizeof(int): Calcula a quantidade de bytes necessária para armazenar 10 inteiros.
O ponteiro vetor armazena o endereço do primeiro elemento da área de memória alocada dinamicamente para 10 inteiros.

```
int* vetor;
vetor = (int*) malloc(10 * sizeof(int));
```

Sizeof

O operador sizeof é usado para determinar o tamanho, em bytes, de um tipo de dado ou de uma variável. Ele é importante porque diferentes tipos de dados ocupam diferentes quantidades de memória.

```
sizeof(int); // Retorna o tamanho de um inteiro (geralmente 4 bytes)
sizeof(double); // Retorna o tamanho de um double (geralmente 8 bytes)
```

Sizeof

```c
int main() {
    printf("Tamanho de char: %zu bytes\n", sizeof(char));
    printf("Tamanho de int: %zu bytes\n", sizeof(int));
    printf("Tamanho de float: %zu bytes\n", sizeof(float));
    printf("Tamanho de double: %zu bytes\n", sizeof(double));
    printf("Tamanho de long: %zu bytes\n", sizeof(long));
    printf("Tamanho de long long: %zu bytes\n", sizeof(long long));
    printf("Tamanho de long double: %zu bytes\n", sizeof(long double));
    printf("Tamanho de void*: %zu bytes\n", sizeof(void*));
    return 0;
}
```

```
Tamanho de char: 1 bytes
Tamanho de int: 4 bytes
Tamanho de float: 4 bytes
Tamanho de double: 8 bytes
Tamanho de long: 8 bytes
Tamanho de long long: 8 bytes
Tamanho de long double: 16 bytes
Tamanho de void*: 8 bytes
```

Sizeof

O identificador de formato %zu é usado em C para imprimir valores do tipo **size_t**, que é o tipo de dado retornado pelo operador **sizeof**.

Por que não usar %d ou outros?

%d: Usado para imprimir inteiros do tipo int, que podem ter um tamanho fixo de 4 bytes em sistemas de 32 bits e 64 bits. No entanto, size_t pode ser maior que int, especialmente em sistemas de 64 bits.

Usar %d para imprimir um valor de size_t pode funcionar em sistemas de 32 bits, mas não é seguro ou garantido em sistemas de 64 bits, pois size_t pode ter 8 bytes. Para garantir a portabilidade e segurança, deve-se usar %zu.

Sizeof

Como sizeof retorna um valor do tipo **size_t**, o identificador de formato correto para imprimir esse valor é **%zu**. Esse especificador de formato é garantido para imprimir corretamente tanto em sistemas de **32 bits** quanto de **64 bits**.

O uso de **%zu** garante que o valor retornado por sizeof seja impresso corretamente em sistemas de **32** e **64 bits**, respeitando o tamanho do tipo **size_t**, que é o tipo retornado por **sizeof**.

Listas lineares encadeadas

Ao longo do tempo, devido à execução de múltiplos programas, alocações e desalocações de memória vão deixando espaços com tamanhos distintos disponíveis. Esse problema é chamado de fragmentação de memória e vai tornando cada vez mais difícil alocar posições contíguas de memória na **heap**.

A alocação encadeada é uma forma de se contornar esse problema, reduzindo a sobrecarga com o gerenciamento de memória.

Listas lineares encadeadas

A memória **heap** é uma região da memória de um programa utilizada para alocação dinâmica. Ao contrário da memória stack (pilha), onde variáveis locais e chamadas de funções são gerenciadas automaticamente pelo compilador, a memória **heap** é gerenciada explicitamente pelo programador, permitindo alocar e liberar blocos de memória conforme necessário.

Listas lineares encadeadas

Alocação Dinâmica: O programador pode solicitar e liberar memória dinamicamente em tempo de execução usando funções como malloc(), calloc(), realloc(), e free() em C.

Gerenciamento Manual: A memória alocada na heap não é automaticamente liberada quando uma função retorna (diferente da stack). Cabe ao programador garantir que a memória seja liberada quando não for mais necessária usando a função free().

Tamanho Flexível: A memória heap é ideal para armazenar estruturas de dados cujo tamanho pode mudar durante a execução do programa (como vetores dinâmicos, listas ligadas, árvores, etc.).

Vida Longa das Variáveis: As variáveis alocadas na heap permanecem ativas até que sejam explicitamente liberadas, independentemente de qual função está sendo executada.

Diferença Heap x Stack

Característica	Heap	Stack
Alocação	Dinâmica, usando malloc(), free()	Automática, feita pelo compilador
Gerenciamento	Controlado manualmente pelo programador	Gerenciado automaticamente
Tempo de vida	Até ser explicitamente liberada	Apenas dentro do escopo da função
Tamanho	Geralmente maior que a stack	Menor, espaço limitado
Velocidade	Mais lenta, devido ao gerenciamento manual	Mais rápida, pois o gerenciamento é automático
Uso	Ideal para estruturas de dados dinâmicas	Ideal para variáveis locais e funções

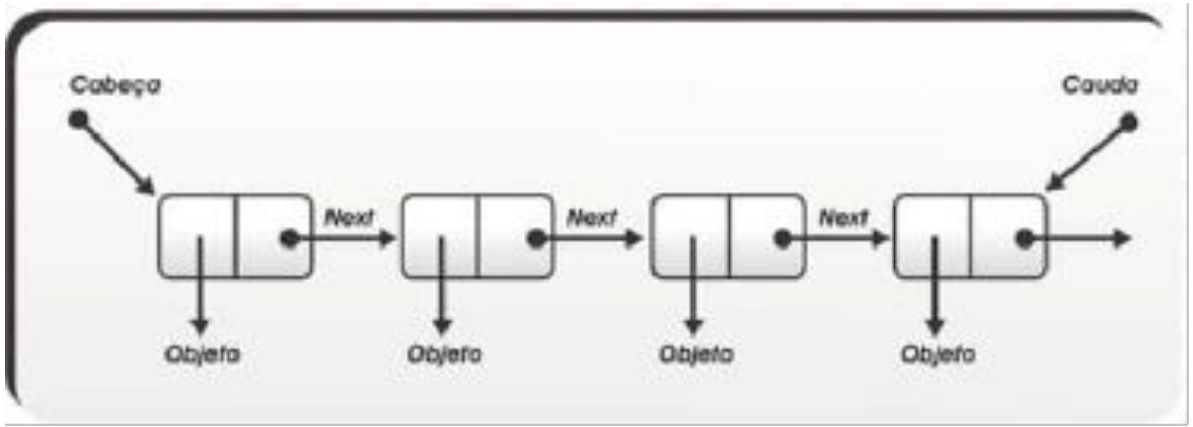
Listas lineares encadeadas

A ideia por trás da alocação encadeada é simplesmente alocar os espaços de memória suficientes para guardar os elementos individualmente e encadeá-los de forma a manter a relação entre eles. Assim, cada elemento da lista ocupará uma posição de memória que pode ou não ser adjacente às demais.

Os elementos na alocação encadeada estão armazenados em posições quaisquer da memória. Logo, não há uma forma de se calcular o endereço dessas posições.

Na alocação encadeada, cada elemento é chamado de nó. Vamos criar um nó especial, chamado “nó cabeça”, cuja finalidade é apenas simplificar as operações sobre a lista.

Listas lineares encadeadas



Lista encadeada

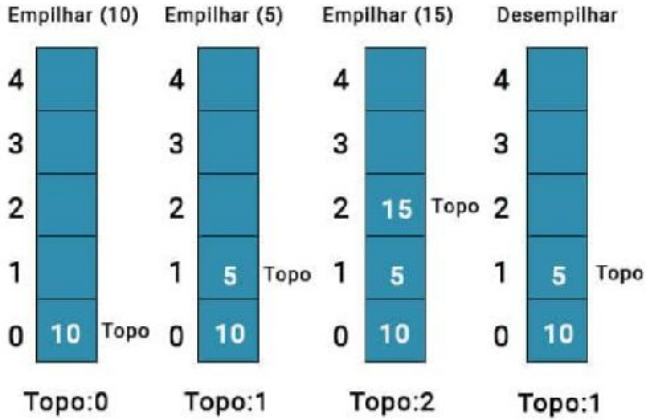
02

Listas, Pilhas e filas

Pilha

Uma pilha é um tipo de lista na qual as operações de inserção, remoção e acesso ocorrem sempre numa mesma extremidade, chamada de topo.

Uma pilha segue a regra “O último a chegar é o primeiro a sair”, também conhecida pela sigla LIFO, do inglês Last In, First Out.

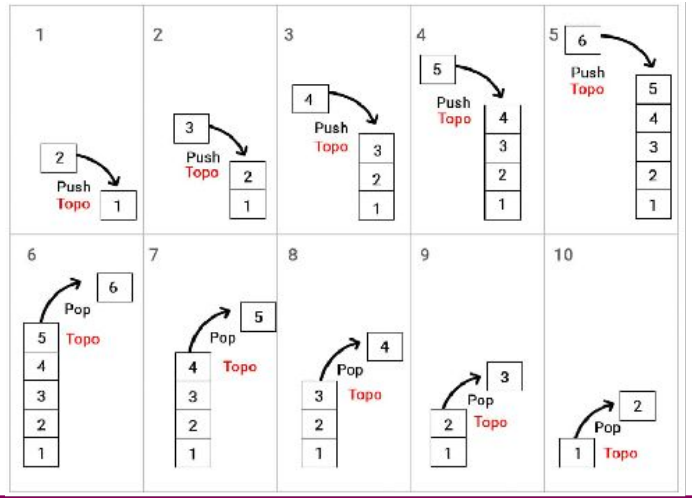


UNIFAVIP wyden

Pilha em alocação sequencial

O operacionalizar uma pilha com alocação sequencial é simples. A operação de inserção (empilhamento) sempre ocorre na extremidade, assim como a remoção (desempilhamento).

Empilhar significa incrementar o topo, e desempilhar, decrementá-lo.



UNIFAVIP wyden

Pilha em alocação sequencial

Em linguagem C, o vetor inicia em 0 (zero), então convencionamos que uma pilha vazia será indicada pelo valor da variável “topo” igual a -1.

Antes de inserirmos um elemento na pilha, precisamos verificar se há posições desocupadas no vetor. Isso é feito comparando-se o valor de “topo” com o tamanho do vetor, o qual é conhecido a priori.

Pilha em alocação sequencial

```c
// Função para inicializar a pilha
void inicializarPilha(Pilha *p) {
    p->topo = -1; // Pilha vazia
}

// Função para verificar se a pilha está cheia
int pilhaCheia(Pilha *p) {
    return p->topo >= MAX - 1; // Se topo é igual ou maior que o tamanho máximo - 1
}

// Função para verificar se a pilha está vazia
int pilhaVazia(Pilha *p) {
    return p->topo == -1; // Se topo é -1, a pilha está vazia
}

// Função para empilhar (inserir) um elemento
void empilhar(Pilha *p, int valor) {
    if (pilhaCheia(p)) {
        printf("Erro: Pilha cheia. Não é possível empilhar %d.\n", valor);
    } else {
        p->dados[++(p->topo)] = valor; // Incrementa topo e insere o valor
        printf("Empilhado: %d\n", valor);
    }
}
```

Pilha em alocação sequencial

```c
// Função para desempilhar (remover) um elemento
int desempilhar(Pilha *p) {
    if (pilhaVazia(p)) {
        printf("Erro: Pilha vazia. Não é possível desempilhar.\n");
        return -1; // Retorna -1 como erro
    } else {
        return p->dados[(p->topo)--]; // Retorna o valor e decrementa topo
    }
}
```

Atividade

Implemente um programa em C, que permite o usuário criar um vetor dinâmico de inteiros, o usuário vai indicar o tamanho do vetor, vai inserir os valores deste vetor e por fim, vão ser impressos na tela. Obs: utilize **malloc**.