

FGV EMAp
Thalis Ambrosim Falqueto

Estrutura de Dados

Resumo

Rio de Janeiro

2025

Sumário

Introdução	3
1. Complexidade de algoritmos	3
1.1 Notação Big O	3
2. Tipos Abstratos de Dados	4
2.1 Pilhas	4
2.2 Filas	6
2.3 Fila circular	8
2.4 Comparação entre TADs	10
2.5 Lista (simplesmente) encadeada	10
2.6 Lista encadeada circular	13
2.7 Lista duplamente encadeada	13
2.8 Comparação entre arrays e listas encadeadas	17
3. Ordenação	18
3.1 - Características relevantes:	18
3.2 Selection Sort	18
img	18
3.3 Insertion Sort	19

Introdução

Olá, basicamente essa é uma tentativa de explicar digitalmente um conteúdo aprendido no terceiro semestre da minha graduação, mais especificamente, o conteúdo aprendido em Estrutura de Dados. Além disso, é meu objetivo aprender a usar a ferramenta do typst. Inicialmente é esperado que você, caro telespectador, tenha algum conhecimento em C++.

Vamos lá!

1. Complexidade de algoritmos

O objetivo principal de calcular a complexidade de um algoritmo é compará-lo com outro para saber qual é o melhor, e assim, escolher o melhor algoritmo possível que resolva seu problema.

Os dois principais tipos de complexidade são:

1. Complexidade espacial: Quantidade de memória necessária para resolver o problema.
2. Complexidade temporal: Quantidade de tempo necessário para resolver o problema.

O foco dado aqui será em calcular a complexidade temporal. Você pode ter pensado que calcular essa complexidade é simples: basta contar quanto tempo um algoritmo leva para resolver um problema e compará-lo com outro algoritmo, certo? Mas isso não é o ideal, pois aí entram outras variáveis, como:

- Especificações do computador
- Linguagem de programação
- Condições de execução

Enfim, todos esses pontos influenciam diretamente no tempo do seu algoritmo, e, portanto, na sua complexidade. Por isso precisamos de algo mais genérico, e a solução para esse problema foi calcular a quantidade de operações que o algoritmo faz, ligando-o diretamente com o tamanho do problema.

1.1 Notação Big O

Para mensurar a quantidade de operações feita num algoritmo, é necessário introduzir a notação de Big O que permanecerá conosco até o fim desse resumo.

Em modos gerais, ela basicamente simplifica o cálculo de toda a função que determina o tempo de execução de um algoritmo pegando o termo de maior grandeza, que determina o pior caso no caso de uma quantidade grande de operações.

Exemplo: Qual a complexidade de execução desse algoritmo?

```
1  #include <iostream>
2
3  float media(float arr[], int n) {
4      float total = 0;
5      for(int i = 0; i < n; i++) {
6          total += arr[i];
7          total = total / i
```

C++

```
8      }  
9      return total;  
10 }
```

Vamos contar a quantidade de operações (ignorando que a definição da função, e contando cada operação de mesmo esforço computacional):

Temos a definição da variável total, somando 1, e, dentro do for, temos outras duas operações feitas (soma e divisão), mas elas dependem do tamanho do array (que tem tamanho n), temos então $2n$ operações ali. Ao fim, damos um return, contando mais 1.

Logo, chamando de $T(n)$ a função que determina a complexidade de execução total do algoritmo, temos: $T(n) = 1 + 2n + 1 = 2n + 2$

Agora, qual seria o pior caso? Aconteceria se n fosse muito grande, certo? Se n fosse muito grande (tendendo a infinito), as constantes 2 que somam e multiplicam a n não importariam o suficiente e, portanto, dizemos que esse algoritmo tem complexidade de execução $O(n)$.

Existem outras análises, que calculam o melhor caso, caso médio, etc. Não vamos entrar nesse assunto no momento, e sim na próxima matéria, PAA.

Por fim, a definição formal da notação big O para encontrar o pior caso é: Dizemos que a função $f(n) = O(g(n))$ se existir uma constante c e um valor n_0 tal que $f(n) \leq cg(n)$.

2. Tipos Abstratos de Dados

TAD é um modelo conceitual de uma estrutura de dados que define um conjunto de operações sem especificar como elas são implementadas. Diferente dos tipos primitivos, que possuem uma representação direta na memória, um TAD é definido em termos de comportamento e operações permitidas, sem impor detalhes sobre sua implementação. Como exemplo temos listas, tuplas, conjuntos, dicionários, filas, pilhas, árvores, etc.

Na prática, para criar um TAD, utilizamos estruturas de dados concretas, como arrays e ponteiros, e definimos funções que respeitam a interface do TAD.

A importância de estudarmos essas Estrutura de Dados é que, basicamente, cada TAD é melhor em resolver um problema específico. Então, de acordo com cada problema, escolhemos o TAD que melhor o soluciona e fazemos uso dele.

Basicamente, temos 3 operações principais: inserir, remover e buscar. Vamos ver alguns TADs e seus desempenhos em cada uma dessas.

2.1 Pilhas

Uma pilha é uma Estrutura de Dados que segue a política LIFO (Last in, First Out), ou seja, o último elemento adicionado da pilha é o primeiro a ser removido.

Algumas aplicações:

1. Controle de chamadas de funções;
2. Fazer/Desfazer (ctrl y/ctrl z);
3. Navegação;

Enfim, vamos analisar a estrutura de dados desse TAD:

```
1 struct Stack {
2     int * data;
3     int maxSize;
4     int top;
5 };
```

Temos um ponteiro que aponta para a pilha, um inteiro que guarda o tamanho máximo da pilha e um inteiro e outro inteiro que aponta para o topo(a quantidade de elementos na pilha atualmente).

Além disso, aqui vai algumas funções básicas dessa Estrutura:

- Inicialização da Pilha - apenas declaramos uma nova pilha, apontamos cada elemento da lista para seu respectivo elemento e a retornamos:

```
1 Stack* initializationStack(int maxSize) {
2     Stack * s = new Stack();
3     s->data = new int[maxSize];
4     s->maxSize = maxSize;
5     s->top = -1;
6     return s;
7 }
```

- Push de um elemento - Se a pilha não estiver cheia, incrementamos a contagem do topo e acessamos o próximo elemento após o topo e colocamos o value lá:

```
1 int pushStack(Stack *s, int value) {
2     if (s->top == s->maxSize - 1) {
3         return 0; // Full stack
4     }
5     s->top += 1;
6     s->data[s->top] = value;
7     return 1;
8 }
```

- Remoção de um elemento - Caso a pilha não esteja vazia, acessamos o ponteiro value usando o último elemento da pilha e decrementamos da contagem de topo:

```
1 int popStack(Stack *s, int *value) {
2     if (s->top == -1) {
3         return 0; // Empty stack
4     }
5     *value = s->data[s->top];
6     s->top -= 1;
7     return 1;
8 }
```

```
8 }
```

- Busca de um elemento - Se a pilha não estiver vazia, apenas retornamos o último elemento colocado da lista:

```
1 int peekStack(const Stack *s, int *value) {  
2     if (s->top == -1){  
3         return 0; // Empty stack  
4     }  
5     *value = s->data[s->top];  
6     return 1;  
7 }
```

- Destruição da pilha - Deleta o array e a pilha:

```
1 void destroyStack(Stack* s) {  
2     delete[] s->data;  
3     delete s;  
4 }
```

O básico é isso, terão coisas mais legais nos exercícios.

2.2 Filas

Uma fila é um tipo de Estrutura de Dados que segue a política FIFO(First IN, First Out), ou seja, o primeiro elemento a ser adicionado na fila é também o primeiro a ser removido.

Algumas aplicações:

1. Assistir mais tarde (YouTube);
2. Gerenciamento de entrada em eventos;
3. Sistema de atendimento de Bancos;

Enfim, vamos analisar a estrutura desse TAD:

```
1 struct Queue {  
2     int* data;  
3     int maxSize;  
4     int size;  
5 };
```

Análogo a pilha, a estrutura contém um ponteiro para o array, um inteiro que guarda o tamanho máximo pilha e outro inteiro que guarda o tamanho atual da pilha.

Ok, vamos para as funções básicas dessa estrutura:

- Inicialização da fila - Apenas inicia uma nova fila e declara cada variável com seu respectivo valor, e, por fim, retorna a fila.

```
1 Queue * initializationQueue(int maxSize) {
```

```

2   Queue * q = new Queue();
3   q->data = new int[maxSize];
4   q->maxSize = maxSize;
5   q->size = 0;
6   return q;
7 }

```

- Push de um elemento - Se a fila estiver cheia, não há nada mais a fazer, caso contrário, adicionamos no fim da fila o valor value e incrementamos a quantidade de elementos na lista.

```

1 int pushQueue(Queue *q, int value){
2     if (q->size == q->maxSize) {
3         return 0; // Full queue
4     }
5     q->data[q->size] = value;
6     q->size += 1;
7     return 1;
8 }

```

Remoção de um elemento - Caso a fila não esteja vazia, salva o primeiro valor adicionado na fila atual, e depois abrimos um for para mover cada item da fila um elemento atrás. Por fim, decrementa o tamanho da fila.

```

1 int popQueue(Queue *q, int *value) {
2     if (q->size == 0) {
3         return 0; // Empty queue
4     }
5
6     *value = q->data[0];
7     for (int i = 1; i < q->size; i++) {
8         q->data[i - 1] = q->data[i];
9     }
10    q->size -= 1;
11    return 1;
12 }

```

- Busca de um elemento - Caso a fila não esteja vazia, apenas seleciona o primeiro elemento adicionado a fila atual e o aloca na variável value.

```

1 int peekQueue(const Queue *q, int *value) {
2     if (q->size == 0) {
3         return 0; // Empty queue
4     }
5     *value = q->data[0];

```

```
6     return 1;
7 }
```

- Destruição da fila - Destrói o array e depois a fila.

```
1 void destroy(Queue* q) {
2     delete[] q->data;
3     delete q;
4 }
```

C++

2.3 Fila circular

Analizando a função popQueue da fila comum, notamos que existe um for para a realocação dos elementos da fila. Isso causa um problema de uma complexidade de execução de $O(n)$. Para contornar isso, uma boa ideia é a implementação de uma fila circular, com head(cabeça/ponta) e tail(cauda/fim).

As aplicações são as mesmas, portanto, vamos para a estrutura desse TAD:

```
1 struct CircularQueue {
2     int *data;
3     int maxSize;
4     int size;
5     int head;
6     int tail;
7 };
```

C++

A estrutura é semelhante à fila, a menos de que, agora, temos dois novos inteiros que marcam os índices do head e do tail da fila.

Vamos analisar o que mudam as funções básicas da fila circular:

- Inicialização da fila circular - Análogo a fila, apenas declara as outras variáveis adicionadas na estrutura.

```
1 CircularQueue * initializationCircularQueue(int maxSize) {
2     CircularQueue * cq = new CircularQueue();
3     cq->data = new int[maxSize];
4     cq->head = 0;
5     cq->tail = -1;
6     cq->size = 0;
7 }
```

C++

- Push de um elemento - Aqui a diferença é que temos que atualizar o tail da fila. Para isso, incrementamos o tail, pois adicionamos algo ao final da fila, e tiramos o módulo em relação ao tamanho da lista, para que o tail “resete” quando o array da fila circular chega ao fim.


```

1 int pushCircularQueue(CircularQueue *cq, int value) {
2     if (cq->size == cq->maxSize) {
3         return 0; // Full circular queue
4     }
5     cq->tail = (cq->tail + 1) % cq->maxSize;
6     cq->data[cq->tail] = value;
7     cq->size++;
8     return 1;
9 }

```

- Remoção de um elemento - Agora, não precisamos mais do for, e após salvar o valor no ponteiro de value, atualizamos a cabeça da cauda. Como queremos tirar(não tiramos, já que não deletamos o valor) o primeiro valor da fila, apenas incrementamos o head(usando o mesmo argumento do módulo no push), passando assim a apontar a head para o próximo valor imediatamente após o head antigo.

```

1 int popCircularQueue(CircularQueue *cq, int *value) {
2     if (cq->size == 0) {
3         return 0; // Empty circular queue
4     }
5     *value = cq->data[cq->head];
6     cq->head = (cq->head + 1) % cq->maxSize;
7     cq->size--;
8     return 1;
9 }

```

- Busca de um elemento - Nada para mudar aqui.

```

1 int peekCircularQueue(CircularQueue *cq, int *value) {
2     if (cq->size == 0) {
3         return 0; // Empty circular queue
4     }
5     *value = cq->data[cq->head];
6     return 1;
7 }

```

- Destruição da fila - Nada para mudar aqui.

```

1 void destroyCircularQueue(CircularQueue *cq) {
2     delete[] cq->data;
3     delete cq;
4 }

```

2.4 Comparação entre TADs

Para cada propósito ao qual cada TAD é designado, quase todas as principais operações são $O(1)$. Veja:

Operação	Pilha	Fila com array	Fila com array circular
inserir	$O(1)$	$O(1)$	$O(1)$
Remover	$O(1)$	$O(n)$	$O(1)$
Buscar	$O(1)$	$O(1)$	$O(1)$

Mas, até agora, todas as estruturas de dados que aprendemos têm em geral, um problema: Todas elas são baseadas num array de tamanho fixo, passado na inicialização do TAD. Isso é um problema por dois motivos:

1. Excesso de espaço dependendo do caso;
2. Falta de espaço dependendo do caso.

Como solucionar esse problema?

2.5 Lista (simplesmente) encadeada

Uma lista encadeada é uma lista que usa ponteiros que indicam o próximo elemento da lista, sem depender uma estrutura de dados pronta que define o tamanho da lista na inicialização.

Algumas aplicações:

1. As mesmas de filas e pilhas(podem ser feitas com listas encadeadas);
2. Tabelas Hash;

Vamos pra estrutura!

```
1 struct Node {
2     Node* next;
3     int value;
4 };
5
6 struct SingleLinkedList {
7     Node* head;
8     int size;
9 };
```

Note que agora são necessárias duas estruturas, uma para o nó, onde temos um inteiro que armazena o valor do nó(tal qual um elemento de um array) e o ponteiro para o próximo elemento, e outra para administrar a lista em si, que controla o ponteiro inicial, e um inteiro que informa o tamanho da lista.

Agora que você já está mais familiarizado com a aparência das funções(verificações básicas, interações com ponteiros), vou deixar de comentar algumas parte do código, já que são mais simples e, além disso, as funções acabam ficando maiores. Vamos as funções básicas:

- Inicialização da lista encadeada - Pela primeira vez, vemos uma atribuição de nullptr, que, como diz o próprio nome, aponta para um ponteiro nulo(todos os nullptr do mesmo código apontam para o mesmo local).

```
1 SingleLinkedList* InitializationSLList() {
2     SingleLinkedList* list = new SingleLinkedList;
3     list->head = nullptr;
4     list->size = 0;
5     return list;
6 }
```

- Push de um elemento:
 - Front - Como queremos adicionar na frente, apenas declaramos um novo nó e fazemos ele apontar para o head da lista, após isso atualizamos o head para ser o novo nó e incrementamos o tamanho.
 - End - Agora o next do novo nó será vazio, e se a lista não for vazia, criamos um nó temporário que serve para andar pela lista, começando pelo head. Enquanto next do nó temporário não for nullptr, significa que não chegamos ao final, e portanto, a lista não chegou ao fim. Quando chegamos ao ponteiro no fim da lista, então colocamos o next dessa lista como o nó criado

```
1 void pushFrontSLList(SingleLinkedList* list, int value) {
2     Node* newNode = new Node;
3     newNode->value = value;
4     newNode->next = list->head;
5     list->head = newNode;
6     list->size++;
7 }
8
9 void pushEndSLList(SingleLinkedList* list, int value) {
10    Node* newNode = new Node;
11    newNode->value = value;
12    newNode->next = nullptr;
13    if (list->head == nullptr) {
14        list->head = newNode;
15    } else {
16        Node* temp = list->head;
17        while(temp->next != nullptr) {
18            temp = temp->next;
19        }
20        temp->next = newNode;
21    }
22    list->size++;
23 }
```

- Remoção de um elemento:

- Front - Nesse caso, sequer precisamos saber o valor a ser deletado, já que será o elemento do início da lista. Salvamos o primeiro elemento da lista, e dizemos que agora a head da lista é o próximo elemento depois do head, e agora podemos deletar o elemento antes salvo.
- Middle/end - Agora precisamos do valor a ser deletado, e após salvarmos o ponteiro atual da lista, fazemos um while que deve encontrar o valor exato, e, para isso, devemos verificar se o next do ponteiro é nulo e se o valor do ponteiro é exatamente o valor procurado. Se entrarmos no próximo if, significa que parou na primeira condição do while e, portanto, o ponteiro é nulo. Caso contrário, é porque encontramos o nó de valor desejado, portanto salvamos o ponteiro, atualizamos o current e deletamos o nó do valor.

```
1 void popFrontSLList(SingleLinkedList* list) {
2     if (list->head == nullptr) {
3         return -1; // Empty queue
4     }
5     Node* temp = list->head;
6     list->head = list->head->next;
7     delete temp;
8     list->size--;
9 }
10
11 void popEndSLList(SingleLinkedList* list, int value) {
12     if (list->head == nullptr) {
13         return;
14     }
15     Node* current = list->head;
16     while (current->next != nullptr && current->next->value != value) {
17         current = current->next;
18     }
19     if (current->next == nullptr) {
20         return;
21     }
22     Node* temp = current->next;
23     current->next = current->next->next;
24     delete temp;
25     list->size--;
26 }
```

Note que aqui temos o popEnd seria análogo ao popMiddle, já que, de qualquer forma, teríamos que percorrer a lista inteira para encontrar o elemento, dado que provavelmente não sabemos onde ele está.

- Busca de um elemento - Percorre toda a lista usando um ponteiro auxiliar até achar o valor, retornando true. Caso não achar, retorna false.

```
1  bool searchSLList(SingleLinkedList* list, int value) {
2      Node* current = list->head;
3      while (current != nullptr) {
4          if (current->value == value) {
5              return true;
6          }
7          current = current->next;
8      }
9      return false;
10 }
```

- Destruição da lista - Em geral isso não é muito útil, já que precisamos passar elemento a elemento para deletar, e que essa lista não usa uma estrutura pronta como array.

```
1  void deleteSLList(SingleLinkedList* list) {
2      Node* current = list->head;
3      while (current != nullptr) {
4          Node* temp = current;
5          current = current->next;
6          delete temp;
7      }
8      delete list;
9  }
```

2.6 Lista encadeada circular

Nesse caso, basta colocar um ponteiro tail na struct SingleLinkedList apontando para a cauda (tail), e atualizar nas funções mostradas anteriormente. Como a lista não tem um tamanho pré-fixado, também não é necessário tratar casos com o módulo da lista, etc. Fica a cargo do leitor.

• Obs.1:

Uma limitação da lista encadeada simples é que ela só vê o próximo elemento, o que é problemático em situações como remoção de um nó ou verificação do elemento anterior numa lista. Para solucionar esse problema, basta adicionarmos um ponteiro tail em cada um dos nós da lista!

2.7 Lista duplamente encadeada

Sabendo que você provavelmente já entendeu o conceito, vamos às aplicações:

1. Todas as outras anteriores;
2. Botões de “avançar” e “voltar” num site;
3. Músicas/filmes de uma playlist.

Legal, vamos para sua estrutura:

```
1  struct Node {
2      int value;
3      Node* next;
4      Node* prev;
5  };
6
7  struct DoubleLinkedList {
8      Node* head;
9      Node* tail;
10     int size;
11 };
```

Bastante semelhante a lista simplesmente encadeada, a menos do ponteiro representando o anterior no nó(chamado de prev) e na lista(chamado de tail).

- Inicialização da lista duplamente encadeada - Análogo a simplesmente encadeada, a menos do tail.

```
1 DoubleLinkedList* InitializationDLList() {
2     DoubleLinkedList* list = new DoubleLinkedList;
3     list->head = nullptr;
4     list->tail = nullptr;
5     list->size = 0;
6     return list;
7 }
```

- Push de um elemento:
 - Front - Aqui, quando verificamos se a lista não é vazia, como queremos atualizar na frente, apontamos o previous do head para o nó a ser adicionado, e depois atualizamos o head da lista como o novo nó. Depois verificamos se ela está vazia e, se estiver, após adicionar o nó como head, colocamos o nó como tail e incrementamos o size.
 - End - Criamos o novo nó, verificamos se a lista é nula e, caso for, colocamos ele como o head. Após isso, verificamos se o tail não é nulo, ou seja, se a lista não é vazia. Caso não seja, então o tail da lista aponta para o nó adicionado. Atualizamos o tail após isso e incrementamos o size.

```
1 void pushFrontDLList(DoubleLinkedList* list, int value) {
2     Node* newNode = new Node{};
3     newNode->value = value;
4     newNode->next = list->head;
5     newNode->prev = nullptr;
6     if (list->head != nullptr) {
7         list->head->prev = newNode;
8     }
```

```

9     list->head = newNode;
10    if (list->tail == nullptr) {
11        list->tail = newNode;
12    }
13    list->size++;
14 }
15
16 void pushEndDLList(DoubleLinkedList* list, int value) {
17     Node* newNode = new Node{};
18     newNode->value = value;
19     newNode->next = nullptr;
20     newNode->prev = list->tail;
21
22     if (list->head == nullptr) {
23         list->head = newNode;
24     }
25     if (list->tail != nullptr) {
26         list->tail->next = newNode;
27     }
28     list->tail = newNode;
29     list->size++;
30 }

```


- Remoção de um elemento:

- ▶ Front - Após verificarmos se a lista não esta vazia, criamos um ponteiro temporário que salvará o ponteiro a ser deletado, logo após atualizamos o head da lista, e com ela atualizada, verificamos se o head atual não é nulo, pois caso ele não seja, devemos atualizar também o prev da nova head. Caso ele seja nulo, então o tail também deve ser atualizado como nulo. Deletamos o ponteiro antigo e decrementamos o size.
- ▶ End - Fazemos literalmente a mesma coisa, só que olhando para o tail. Note como a estrutura é parecida

```

1 void popFrontDLList(DoubleLinkedList* list) {
2     if (list->head == nullptr) {
3         return;
4     }
5     Node* temp = list->head;
6     list->head = list->head->next;
7     if (list->head != nullptr) {
8         list->head->prev = nullptr;
9     } else {
10        list->tail = nullptr;
11    }

```

 C++

```

12     delete temp;
13     list->size--;
14 }
15
16 void popEndDLList(DoubleLinkedList* list) {
17     if (list->tail == nullptr) {
18         return;
19     }
20
21     Node* temp = list->tail;
22     list->tail = list->tail->prev;
23     if (list->tail != nullptr) {
24         list->tail->next = nullptr;
25     } else {
26         list->head = nullptr;
27     }
28     delete temp;
29     list->size--;
30 }

```

Note que não colocamos as funções de push e pop no meio da lista, já que isso quebraria a ideia dessas operações serem $O(1)$, pois teríamos que percorrer toda a lista até encontrar o elemento do meio, se fosse o caso. Além disso, um push no meio não faz sentido, seria algo como furar fila, enquanto o pop no meio de uma fila pode realmente acontecer, como uma desistência. Aqui vai o código do pop no meio da lista:

```

1 void popMiddleDLList(DoubleLinkedList* list, int value) {
2     if (list->head == nullptr) {
3         return;
4     }
5     Node* current = list->head;
6     while (current != nullptr && current->value != value) {
7         current = current->next;
8     }
9     if (current == nullptr) {
10        return;
11    }
12    current->prev->next = current->next;
13    if (current->next != nullptr) {
14        current->next->prev = current->prev;
15    }
16    delete current;
17    list->size--;

```



```
18 }
```

- Busca de um elemento - Análogo a lista simplesmente encadeada.

```
1  bool searchDLList(DoubleLinkedList* list, int value) {
2      Node* current = list->head;
3      while (current != nullptr) {
4          if (current->value == value) {
5              return true;
6          }
7          current = current->next;
8      }
9      return false;
10 }
```

- Destruição da lista - Análogo a lista simplesmente encadeada.

```
1  void deleteDLList(DoubleLinkedList* list) {
2      Node* current = list->head;
3      while (current != nullptr) {
4          Node* temp = current;
5          current = current->next;
6          delete temp;
7      }
8      delete list;
9  }
```

2.8 Comparação entre arrays e listas encadeadas

Após todo esse código, podemos fazer uma breve comparação entre listas encadeadas e arrays:

- Arrays são uma boa escolha quando há uma estimativa da quantidade de elementos a serem inseridos e permitem acesso rápido a qualquer elemento via índice, mas inserções e remoções no meio são custosas, pois exigem deslocamento de elementos .
- Listas encadeadas são boa escolha quando a quantidade de elementos pode variar significativamente e inserções e remoções são eficientes, pois não exigem deslocamento de elementos, porém, acesso a elementos individuais é mais lento, pois requerem percorrer a lista.

Além disso, embora listas encadeadas sejam flexíveis e eficientes para inserção e remoção, elas possuem algumas desvantagens:

1. Maior uso de memória por elemento (devido aos ponteiros adicionais);
2. Não é possível acessar uma posição aleatória da lista de forma eficiente;

Uma ideia que possibilita o acesso a uma posição aleatória de forma eficiente e melhora a busca linear é ordenar a lista. Mas como fazer isso?

3. Ordenação

Como introduzido, por vezes gostaríamos que nossa Estrutura de Dados estivesse ordenada, pois a ordem dos elementos pode ser mais importante que a inserção ou remoção. Por exemplo, num sistema de e-commerce, onde os produtos são identificados por preço, ou até num hospital, onde gostaríamos de ordenar por prioridade.

Ordenar os dados é essencial, pois:

1. permite buscas mais eficientes;
2. define uma ordem de prioridade;
3. facilita a análise estatística(Ex: mediana);

3.1 - Características relevantes:

A utilidade dos algoritmos de ordenação que vamos ver podem ser medidos através de:

- Complexidade de tempo de execução;
- Complexidade de espaço utilizado: quantidade de espaço adicional de memória necessária (além do array de entrada);
- Estabilidade: se mantém a ordem relativa dos elementos iguais na entrada;
 - Exemplo: No caso do exemplo do hospital, se cada elemento(pessoa) tem uma prioridade, é esperado que pessoas de mesma prioridade continuem na mesma ordem que chegaram. Portanto, ao ordenar pela prioridade, o algoritmo é estável se cada elemento de mesma prioridade se mantém na mesma ordem antes de ordenar.
- In-place vs Out-of-place:
 - In-place: Não requer memória extra significativa.
 - Out-of-place: Requer uma estrutura auxiliar para armazenar os elementos ordenados;
- Performance em diferentes tamanhos de entrada: Alguns algoritmos podem ser melhores que outros para quantidades pequenas ou grandes de dados.

Existem outros tipos de características relevantes, como adaptabilidade e paralelização, mas não serão abordados aqui. Legal, vamos para os algoritmos!


3.2 Selection Sort

• Ideia

- Percorre a lista até encontrar o menor elemento;
- Troca esse elemento com o primeiro da lista;
- Repete a ideia para os próximos elementos.

img

Note que precisamos de dois inteiros, um para salvar o índice do menor elemento, e outro para fazer a troca de elementos. O primeiro for passará por toda a lista, e o segundo for comparará os elementos subjacentes ao índice *i*, pois antes desse índice os elementos já foram ordenados. Fazemos a comparação do valor do índice *i* com todos os posteriores, e atualizamos o índice *j*. Após cada comparação, salvamos o valor do menor elemento, atualizamos o valor do índice do menor elemento como o elemento do índice *i*, e por fim atualizamos o valor do índice *i* como o menor elemento.

1	<code>void selectionSort(int arr[], int n) {</code>	// Custo		Vezes	
2	<code>int minIndex, temp;</code>	// 2		1	
3	<code>for (int i = 0; i < n - 1; i++) {</code>	// 2		n-1	
4	<code>minIndex = i;</code>	// 1		n-1	
5	<code>for (int j = i + 1; j < n; j++) {</code>	// 2		n-i+1 -> n-1,	
6	<code>if (arr[j] < arr[minIndex]) {</code>	// 1		n-i+1 -> n-1,	
7	<code>minIndex = j;</code>	// 1		n-i+1 -> n-1,	
8	<code>}</code>				
9	<code>}</code>				
10	<code>temp = arr[minIdx];</code>	// 1		n-1	
11	<code>arr[minIdx] = arr[i];</code>	// 1		n-1	
12	<code>arr[i] = temp;</code>	// 1		n-1	
13	<code>}</code>				
14	<code>}</code>				

- **Características:**

- Complexidade de tempo de execução: $O(n^2)$ para o pior caso, dado dois fors que iteram praticamente até n ;
- Complexidade de espaço: $O(1)$, pois não precisamos criar nada;
- Estabilidade: não é estável, trocas alteram a ordem de elementos iguais.

3.3 Insertion Sort