Trenzao

Os arquivos em OO

▼ Produto.h

```
class Produto {
    private:
      int codigo;
      float preco;
      float custo;
      int estoque;

    public:
        Produto();
        Produto(const Produto &outro);

        ///metodo Getters e Setters
        void setPreco(double preco);
        double getPreco();
};
```

▼ Produto.cpp

```
#include"Produto.h"

void Produto::Produto(){
    this->codigo = -1;
    this->preco = 0.0;
    this->custo = 0.0;
    this->estoque = 0.0;
}

void Produto::setPreco(float preco){
    this->preco = preco;
}

float Produto::getPreco(){
    return this->preco;
}
```

Encapsulamento

- É onde voce esconde o arquivo
- É a maneira onde acessar os atributos da classe .h
- Serve para ocultar detalhes de implementação de u objeto pra outro
- Os atributos são private e os metodos são public
- Os atributos são acessados pelos metodos get(pega) set(atualiza)

Métodos essenciais

- Construtor, construtor de copia, preencher e imprimir
- O construtor é o mais importante pq sem ele a classe não pode ser criada e nem utilizada.
- O construtor de copia é chamado toda vez q um objeto é passado como parametro

Leitura e escrita em arquivo txt

- Faz uso de uma nova biblioteca a <fstream>

```cpp
#include<fstream>
using namespace std;

int main(){
    string caminhoPasta = "teste.txt";
    ofstream arquivoSalvo;
    arquivoSalvo.open(caminhoPasta.c_str());

    arquivoSalvo << "Ola mundo " << endl;

    arquivoSalvo.close();
    return 0;
}
```

Salvar no arquivo

```cpp
#include<fstream>
using namespace std;

int main(){
    string caminhoPasta = "teste.txt";
    ifstream arquivoLido;
    arquivoLido.open(caminhoPasta.c_str());

    float idade;
    arquivoLido >> idade;

    arquivoLido.close();
    return 0;
}
```

Ler no arquivo

Lista contigua

- A classe lista contigua tem 3 atributos: int tam, int quant, Prodto *lista;
- São criados novos metodos: shift-end; shift-front;
- São criados dois metodos no private: temEspaco() e isEmpty(). Elas são criadas no private pq eles so faz sentido nessa classe;

Shift-end

```
1 void ListaProduto::shiftEnd(int ate) {
2    if (temEspaco() && !isEmpty()) {
3        for (int i = quant; i > ate; i--) {
4            lista[i].copiar(lista[i - 1]); //lista[i] = lista[i-1];
5        }
6    }
7 }
```

Shift-front

```
1  void ListaProduto::shiftFront(int aPartir) {
2    if (indiceValido(aPartir)) {
3        if (!isEmpty()) {
4            for (int i = aPartir; i < quant - 1; i++) {
5                lista[i].copiar(lista[i + 1]); //lista[i] = lista[i+1];
6            }
7        }
8    }
9  }
```

Insert (int posicao)

```
1 void ListaContigua::insert(int posicao) {
2    if (!temEspaco()){
3        cout << "Operacao invalida. Lista cheia." << endl;
4    }else if(!indiceValido(posicao) || posicao > quant) {
5        cout << "Operacao invalida para a posicao escolhida." << endl;
6    }else{//tudo ok, pode inserir!
7        shiftEnd(posicao);
8        lista[posicao].preencher();
9        quant++;
10    }
11 }
```

Remove(int posicao)

```
1 void ListaContigua::remove(int posicao) {
2    if(isEmpty()){
3        cout << "Lista vazia" << endl;
4    }else if(!indiceValido(posicao) || posicao >= quant){
5        cout << "Operacao invalida para posicao escolhida.";
6    }else{//tudo ok
7        shiftFront(posicao);
8        quant--;
9    }
10 }
```

Busca em lista contigua

- Lista contigua é otimo pra busca
- Existe dois principais metodos: forca bruta e busca Binaria
- Forca bruta é facil pra implementar mais é ineficiente
- Busca binaria já é dificil de implementar mais é muito eficiente
- Busca binaria tem um lado ruim, a lista tem q ta ordenada

ForcaBruta()

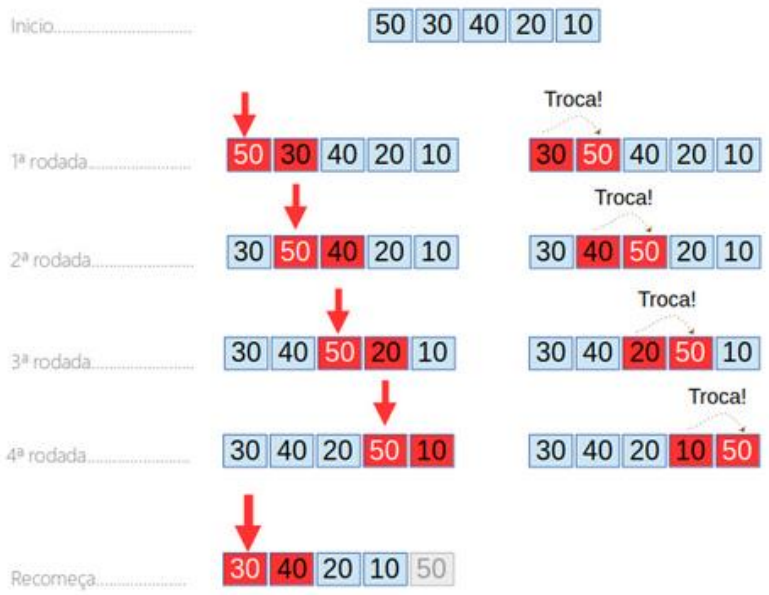
```
//Força bruta
int ListaProduto::buscaForcaBruta(int codigoProcurado) {
    contadorDeComparacoes = 0;
    for (int i = 0; i <= quant - 1; i++) {
        if (lista[i].getCodigo() == codigoProcurado)
            return i;
        else
            contadorDeComparacoes++;
    }
    return -1;
}
```

buscaBinaria()

```
int ListaProduto::buscaBinaria(int codigoProcurado) {
    contadorDeComparacoes = 0;
    int inicio = 0;
    int fim = tam - 1;
    int meio;
    while (inicio <= fim) {
        meio = (inicio + fim) / 2;
        contadorDeComparacoes++;

        if (lista[meio].getCodigo() == codigoProcurado)
            return meio;
        else if (codigoProcurado < lista[meio].getCodigo()) {
            fim = meio - 1;
        } else {
            inicio = meio + 1;
        }
    }
    return -1;
}
```

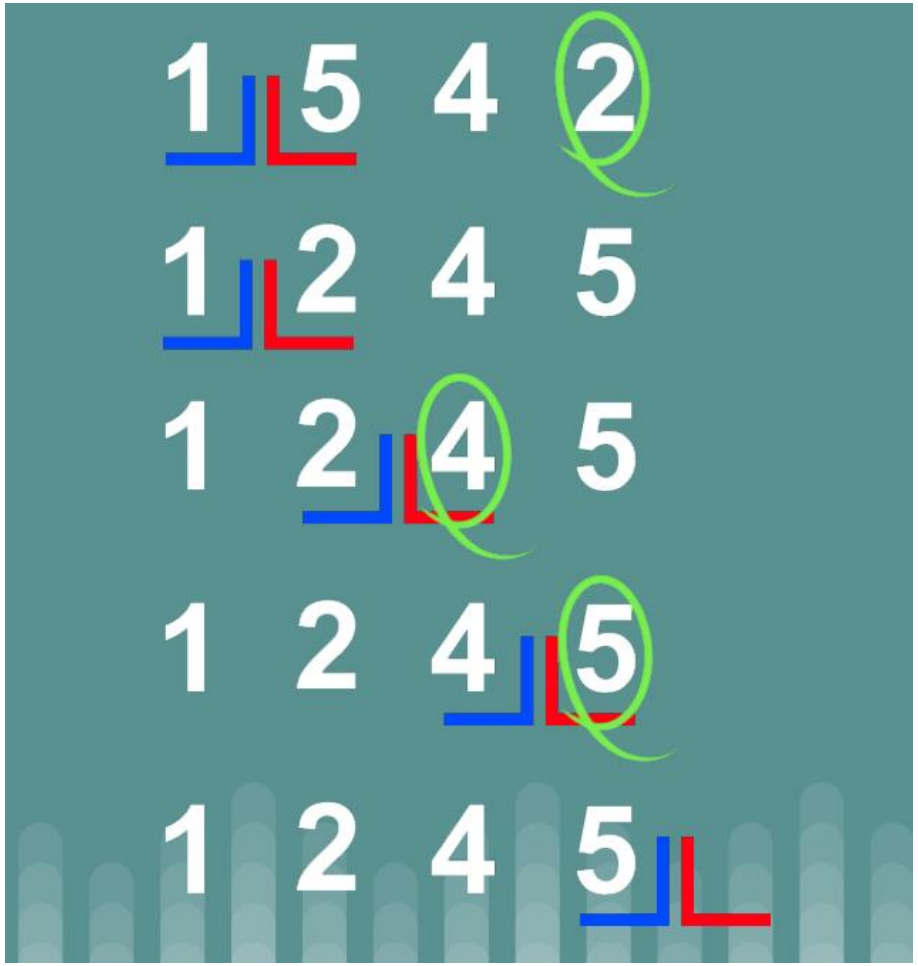
Metodos de ordenação: Bubble Sort

- Facil de implemetar e entender
- Vai comparando um a um até que ordene




```
int Lista::bolha(int elemento){
    int indexFinal = quant-1;

    while(indexFinal > 0){
        for(int i=0; i<= indexFinal-1; i++){
            if(lista[i] > lista[i+1]){
                troca(lista[i], lista[i+1]);
            }
        }
        indexFinal--;
    }
}
```

Selection Sort

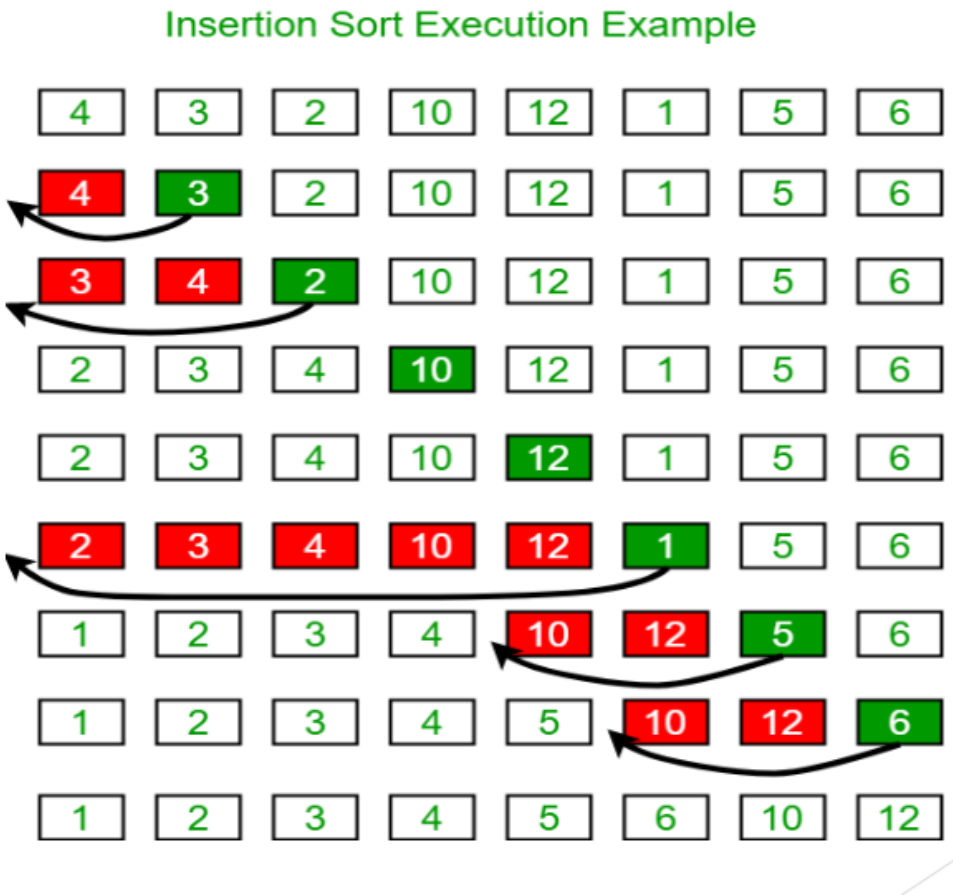



```cpp
#include <iostream>
#include <vector>
using namespace std;

void selectionSort(vector<int> &vec) {
    for (int i = 0; i < vec.size() - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < vec.size(); j++) {
            if (vec[j] < vec[minIndex])
                minIndex = j;
        }
        swap(vec[i], vec[minIndex]);
    }
}

int main() {
    vector<int> vec = {43, 5, 123, 94, 359, -23, 2, -1};
    selectionSort(vec);
    for (int i : vec) {
        cout << i << " ";
    }
    return 0;
}

```

1.Ele começa procurando o menor elemento no vetor e o coloca na primeira posição.
2.Em seguida, procura o segundo menor elemento e o coloca na segunda posição.
3.Este processo continua até que todos os elementos estejam em sua posição correta, ou seja, o vetor esteja ordenado.

Insertion Sort

Ele construiu uma lista ordenada um elemento de cada vez, inserindo cada elemento na posição correta.



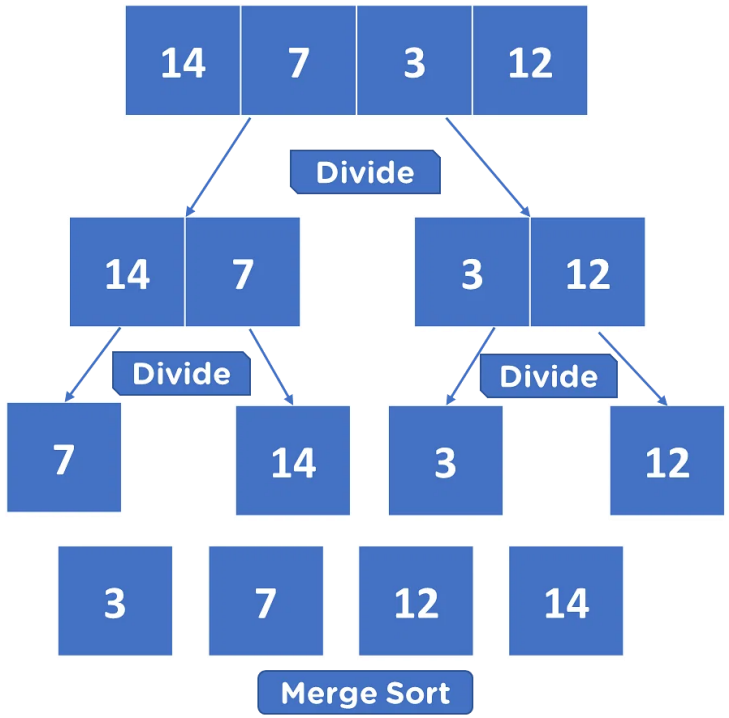

```c
void insertionSort(int arr[], int n) {
   int key, j;
   for (int i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

Merge Sort

```cpp
void merge(int* vetor, int inicio, int meio, int fim){
    int tamEsquerda = meio - inicio +1;
    int tamDireita = fim - meio;
    int* esquerda = new int[tamEsquerda];
    int* direita = new int[tamDireita];

    for(int i = 0; i < tamEsquerda; i++)
        esquerda[i] = vetor[inicio +i];
    for(int j = 0; j < tamDireita; j++)
        direita[j] = vetor[meio +1 +j];

    int i = 0, j = 0, k = inicio;

    while(i < tamEsquerda && j < tamDireita){
        if(esquerda[i] <= direita[j]){
            vetor[k] = esquerda[i];
            i++;
        }else{
            vetor[k] = direita[j];
            j++;
        }
        k++;
    }

    while(i < tamEsquerda){
        vetor[k] = esquerda[i];
        i++;
        k++;
    }

    while(j < tamDireita){
        vetor[k] = direita[j];
        j++;
        k++;
    }

}
```

```cpp
void sort(int* vetor, int inicio, int fim){
    if(inicio < fim){
        int meio = inicio + (fim - inicio) / 2;

        sort(vetor, inicio, meio);
        sort(vetor, meio + 1, fim);

        merge(vetor, inicio, meio, fim);
    }
}

int main(){

    int tam = 9;
    int* vetor = new int[tam] {3,6,5,1,9,7,2,8,4};

    sort(vetor, 0, tam -1);

    for(int i = 0; i < tam; i++){
        cout << vetor[i] << " - ";
    }
    cout << "\n\n";


    return 0;
}
```

- Divide pra conquistar
- Muito eficiente

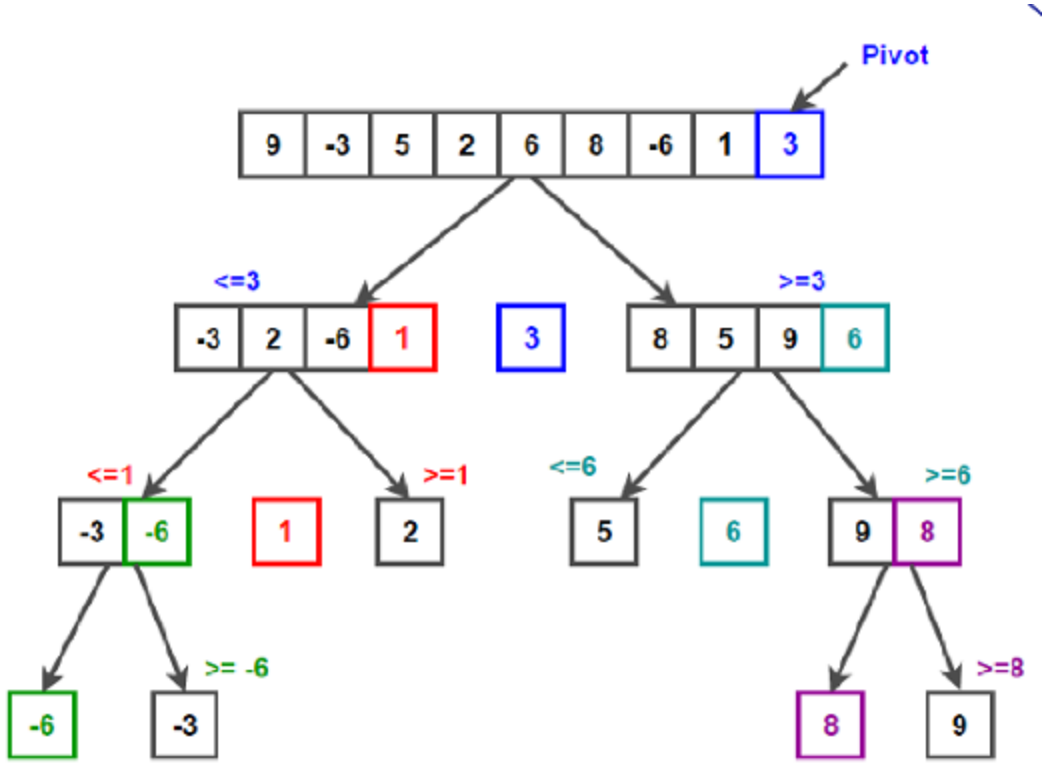


https://github.com/kfrural/MergeSort

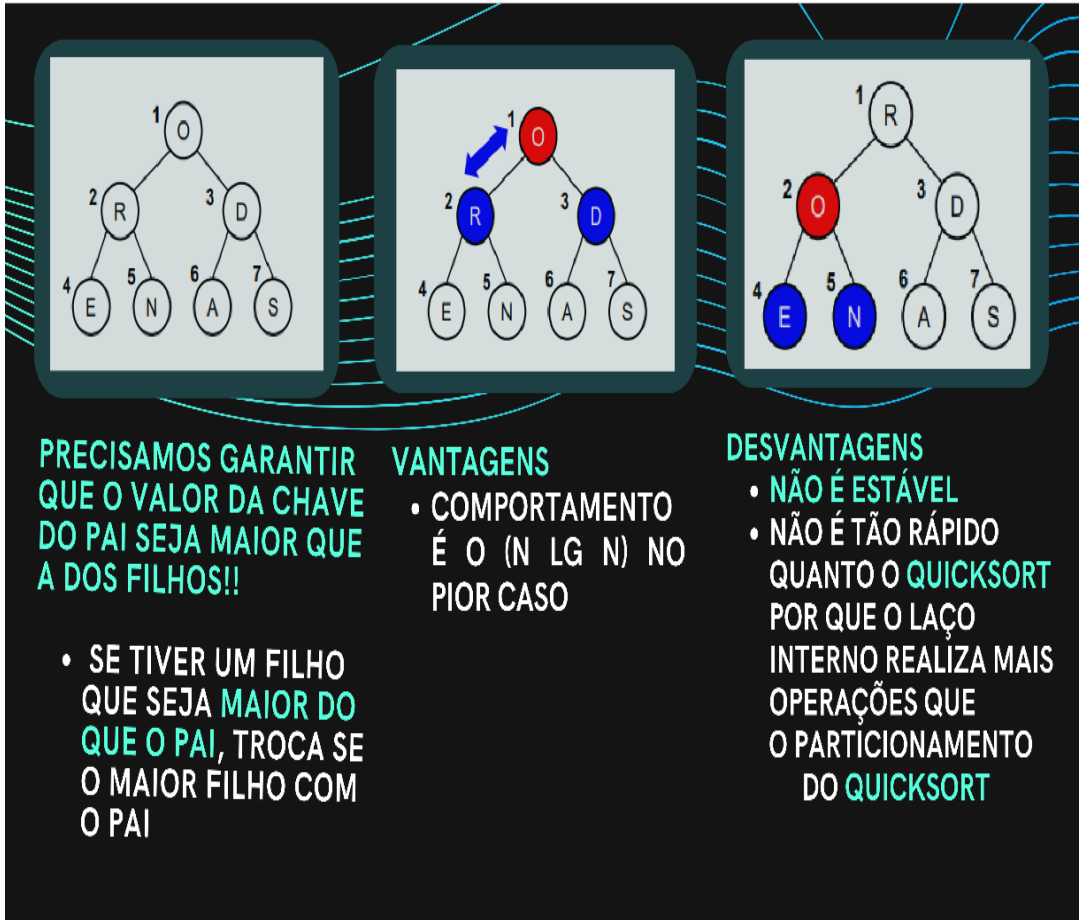
Quick Sort

```
#include <algorithm>

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

```

- É escolhido um pivô.
- A partir desse pivô é ordenado o restante
- De um lado fica os maiores e do outro os menores q o pivô



Heap Sort




```
1   void heapsort(int a[], int n) {
2      int i = n / 2, pai, filho, t;
3      while(true) {
4         if (i > 0) {
5             i--;
6             t = a[i];
7         } else {
8             n--;
9             if (n <= 0) return;
10            t = a[n];
11            a[n] = a[0];
12        }
13        pai = i;
14        filho = i * 2 + 1;
15        while (filho < n) {
16            if ((filho + 1 < n)  &&  (a[filho + 1] > a[filho]))
17                filho++;
18            if (a[filho] > t) {
19               a[pai] = a[filho];
20               pai = filho;
21               filho = pai * 2 + 1;
22            } else {
23               break;
24            }
25        }
26        a[pai] = t;
27     }
28  }
29
```

Lista encadeada

- Não tem elementos sequenciais
- O elemento anterior guarda o endereco do proximo
- A class ListaEncadeada.h tem apenas dois atributos: head e quant
- Quant é a quantiade de elementos e head é um apontador para o primeiro no

Classe ListaEncadeada.h

```
class ListaEncadeada{
    //atributos
    private:
        int quant;
        Nodo* head;

    //métodos
    public:
        ListaEncadeada(); //construtor
    ...
};
```

Class nodo

- A lista encadeada tem por base a classe nodo, mas nodo é independente de lista encadeada

```
class Nodo{
    private:    //atributos
        Produto item;
        Nodo* prox;

    public: //métodos
        Nodo(); //construtor
        Nodo(Produto &p);
        ...
};
```

Metodos insert, remove, getElemento

```
void ListaEncadeada::insert(){
    Produto p;
    p.preencher();
    Nodo* novo = new Nodo(p);

    if(quant == 0){//lista vazia
      head = novo;
    }else{
      novo→setProx(head);
      head = novo;
    }
    quant++;
}
```

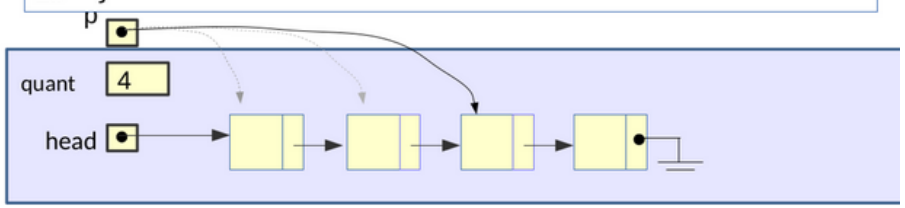
Este método fará a inserção de um novo nó na primeira posição da lista

```
void ListaEncadeada::remove(){
  if(quant==0){
    cout << "Lista vazia. Nada a fazer.";
  }else{
    head = head→getProx();
    quant--;
  }
}
```

Este método faz a **remoção** do **primeiro** elemento da lista

Encadeada.cpp

```
14  Nodo* Encadeada::getElemento(int n){
15     Nodo *p = head;
16     int i = 1;
17     while(i <= n-1 && p→getProx() != NULL){
18        p = p→getProx();
19        i++;
20     }
21     if( i == n)
22        return p;
23     else
24        return NULL;
25  }
```



Este método é responsável por procurar um determinado elemento dentro da lista encadeada e trazer seu endereço

Insert(posicao), remove(posicao)

```
10 void Encadeada::insert(int n) {
11    if (n >= 1 && n <= quant + 1) {
12        Produto p;
13        p.preencher();
14        Nodo* novo = new Nodo();
15        novo->setItem(p);
16
17        if (n == 1) { //estamos no mesmo caso de inserir na 1 posicao
18            novo->setProx(head);
19            head = novo;
20        } else if (n == quant + 1) { //inserindo depois da ultima posicao
21            Nodo* ultimo = this->getElemento(n - 1);
22            novo->setProx(NULL);
23            ultimo->setProx(novo);
24        } else { //inserindo nas posições do meio, caso padrão.
25            Nodo* anterior = this->getElemento(n - 1);
26            novo->setProx(anterior->getProx());
27            anterior->setProx(novo);
28        }
29        quant++;
30    } else {
31        cout << "Operação inválida para esta posição.";
32    }
33 }
```

```
40 void ListaEncadeada::remove(int n){
41    if(n>=1 && n <= quant){
42        if(n == 1){
43          this→remove();
44        }else if(n == quant){
45          Nodo* anterior = this→getElemento(n-1);
46          anterior→setProx(null);
47          quant--;
48        }else{
49          Nodo* anterior = this→getElemento(n-1);
50          Nodo* saira = anterior→getProx();
51          anterior→setProx(saira→getProx());
52          quant--;
53        }
54     }
55 }
```

Lista Duplamente encadeada

- Não tem elementos sequencias
- Temos os elementos para guardar o endereço do anterior e do proximo
- A class ListaDuplamenteEncadeada.h tem apenas dois atributos: head e quant
- Quant é a quantiade de elementos e head é um apontador para o primeiro nó

Nodo.h

```
class Nodo{
    private: //atributos
        Nodo* ant;
        Produto item;
        Nodo* prox;

    public: //métodos
        Nodo(); //construtor
        Nodo(Produto &p);
        ...
};
```

A lista duplamente encadeada tem por base a classe nodo, mas nodo é independente de lista encadeada

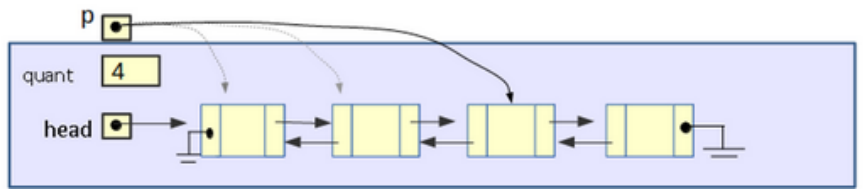
Metodos insert, remove, getElmento

```
void ListaDuplamenteEncadeada::insert(){
    Produto p;
    p.preencher();
    Nodo* novo = new Nodo(p);

    if(quant == 0){//lista vazia
      head = novo;
    }else{
      novo→setProx(head);
      novo→setAnt(NULL);
      head = novo;
    }
    quant++;
}
```

```
void ListaDuplamenteEncadeada::remove(){
  if(quant>0){
    head = head→getProx();
    head→setAnt(NULL);
    quant--;
  }else{
    cout << "Lista vazia. Nada a fazer."<<endl;
  }
}
```

DuploEncadeamento.cpp

```
14  Nodo* DuploEncadeamento::getElemento(int n){
15        Nodo *p = head;
16        int i = 1;
17        while(i <= n-1 && p→getProx() != NULL){
18            p = p→getProx();
19            i++;
20        }
21        if( i == n)
22            return p;
23        else
24            return NULL;
25  }
```



Insert(posicao), remove(posicao)

```
14 void ListaEncadeada::insert(int posicaoN){
15      Produto p;
16      p.preencher();
17      Nodo* novo = new Nodo();
18
19      if(posicaoN <= 1){
20          this→insert();
21      }else if(posicaoN == quant+1){
22          Nodo* n = this.getElemento(posicaoN);
23          n→setProx(novo);
24          novo→setAnt(n);
25          quant++;
26      }else{
27          Nodo* n = this.getElemento(posicaoN);
28          Nodo* anterior = n→getAnt();
29          novo→setProx(n);
30          novo→setAnt(anterior);
31          anterior→setProx(novo);
32          n→setAnt(novo);
33          quant++;
34      }
35  }
```

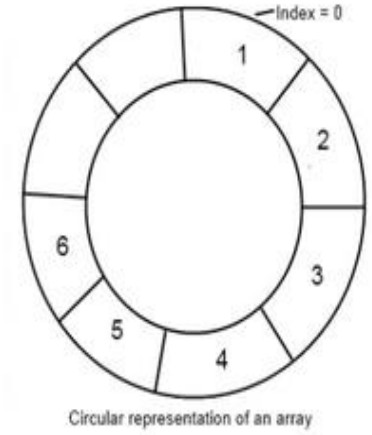
```
25 void DuploEncadeamento::remove(int n){
26      if(n <= 1){
27          this→remove();
28      }else if(n == quant){
29          Nodo* anterior = this.getElemento(n-1);
30          anterior→setProx(null);
31      }else{
32          Nodo* excluido = this.getElemento(n);
33          Nodo* anterior = excluido→getAnt();
34          Nodo* frente = excluido→getProx();
35
36          anterior→setProx(frente);
37          frente→setAnt(anterior);
38          quant--;
39      }
40  }
```

Listas Especiais

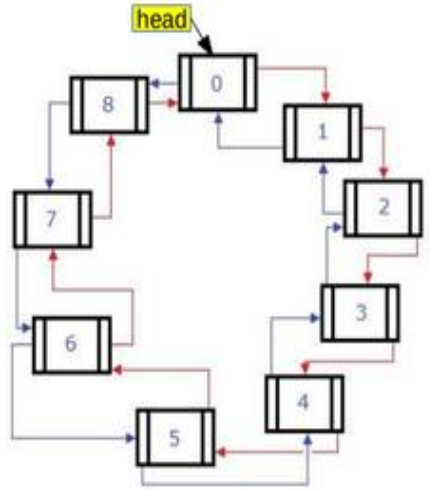
- Tem 3 especificas:
- Lista circular
- Pilha
- fila

Lista Circular

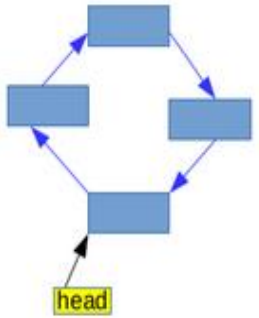
- O sucessor do último é o primeiro, e o antecessor do primeiro é o último



Contígua



Duplamente Encadeada



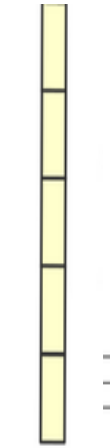
encadeada

Pilha

- Só pode **inserir** no **topo**
- Só pode **remover** no **topo**

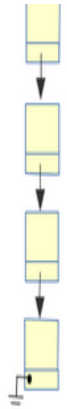
Pilha.h

```
1  class Pilha{
2    private:
3       int tam;
4       int quant;
5       char* pilha;
6    public:
7       Pilha(); //construtor
8       void shiftEnd();
9       void push(char elemento);
10      void insert(char elemento, int indice);
11      void shiftFront();
12      void pop();
13      void remove(int indice);
14      int buscar(char elemento);
15
16 };
```



PilhaEncadeada.h

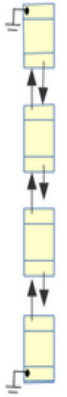
```
1  class PilhaEncadeada{
2    //atributos
3    private:
4       int quant;
5       Nodo* head;
6    //métodos
7    public:
8       PilhaEncadeada(); //construtor
9       void push();
10      void pop();
11      Nodo* getElemento(int n);
11      void insert(int n);
12      void remove(int n);
13      ...
14
15 };
```



- Metodos insert, remove no indice
  - Não faz sentido existir
  - Pois o acesso é sempre no topo

PilhaDuploEncadeamento.h

```
1  class PilhaDuploEncadeamento{
2    //atributos
3    private:
4       int quant;
5       Nodo* head;
6    //métodos
7    public:
8       PilhaDuploEncadeamento(); //construtor
9       void push();
10      void pop();
11      Nodo* getElemento(int n);
11      void insert(int n);
12      void remove(int n);
13      ...
14
15 };
```



Pilha contigua

Pilha Encadeada

Pilha Duplamente

Fila

- Insere no final
- Remove no inicio

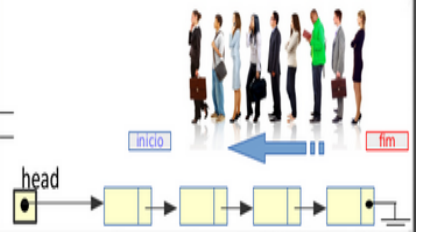
Fila.h

```
1  class Fila{
2    private:
3       int tam;
4       int quant;
5       char* filha;
6    public:
7       Fila(); //construtor
8       void shiftEnd();
9       void insert(char elemento);
10      void insert(char elemento, int indice);
11      void shiftFront();
12      void remove();
13      void remove(int indice);
14      int buscar(char elemento);
15
16 };
```

Fila Contigua

FilaEncadeada.h

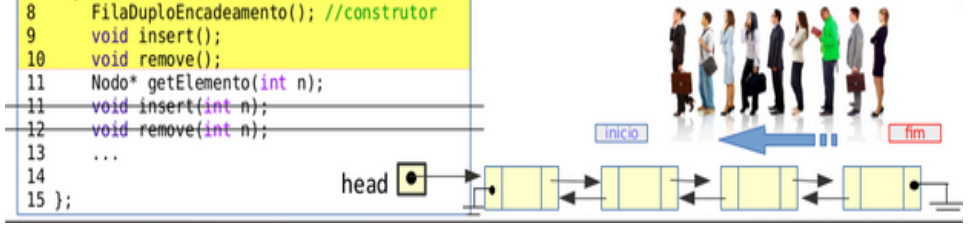
```
1  class FilaEncadeada{
2    //atributos
3    private:
4       int quant;
5       Nodo* head;
6    //métodos
7    public:
8       FilaEncadeada(); //construtor
9       void insert();
10      void remove();
11      Nodo* getElemento(int n);
11      void insert(int n);
12      void remove(int n);
13      ...
14
15 };
```



Fila Encadeada

FilaDuploEncadeamento.h

```
1  class FilaDuploEncadeamento{
2    //atributos
3    private:
4       int quant;
5       Nodo* head;
6    //métodos
7    public:
8       FilaDuploEncadeamento(); //construtor
9       void insert();
10      void remove();
11      Nodo* getElemento(int n);
11      void insert(int n);
12      void remove(int n);
13      ...
14
15 };
```



Fila Duplamente

O método insert() é modificado para que a inserção seja feita sempre na última posição

Lista- slavar e ler arquivo

- Usada para não perder o conteudo ao fechar o programa

```h
class ListaContigua {
  private:
    int tam, quant;
    Cachorro *lista;

    void shiftFront(int aPartir);
    void shiftEnd(int ate);
  public:
    ListaContigua();
    ListaContigua(int tamanho);

    void insert();
    void insert(int posicao);
    void remove();
    void remove(int posicao);

    void loadLista();
    void saveLista();
    void criarListaRandom();

    //GETTERS E SETTERS
    void setTam(int tam);
    int getTam() const;
    ...
};
```

ListaContigua.h

```cpp
#include "ListaContigua.h"

void ListaContigua::saveLista() {
    ContadorTempo temp("Salvando arquivo");

    string caminhoPasta = "teste.txt";
    ofstream arquivoSalvo;
    arquivoSalvo.open(caminhoPasta.c_str());

    //preenchendo o arquivo
    arquivoSalvo << quant << "\n" << tam << "\n";
    for (int i = 0; i <= quant - 1; i++) {
        arquivoSalvo << this->lista[i].getCod()
                << " " << this->lista[i].getNome()
                << " " << this->lista[i].getRaca()
                << " " << this->lista[i].getIdade()
                << " " << this->lista[i].getSexo() << "\n";
    }
    arquivoSalvo << "\n";

    arquivoSalvo.close();
    temp.stop();
}
```

ListaContigua.cpp + saveLista

```cpp
#include "ListaContigua.h"

void ListaContigua::loadLista() {
    string caminhoPasta = "teste.txt";
    ifstream arquivoLido;

    arquivoLido.open(caminhoPasta.c_str());

    int q;
    arquivoLido >> q;
    arquivoLido >> tam;
    this->lista = new Cachorro[this->tam];

    string nome, raca;
    int cod, idade;
    char sexo;
    Cachorro c;
    // while (!arquivoLido.eof()) {
    for (int i = 0; i < q; i++) {
        arquivoLido >> cod;
        arquivoLido >> nome;
        arquivoLido >> raca;
        arquivoLido >> idade;
        arquivoLido >> sexo;
        c.setCod(cod);
        c.setNome(nome);
        c.setRaca(raca);
        c.setIdade(idade);
        c.setSexo(sexo);

        this->insert(c);

    }

    arquivoLido.close();
}
```

ListaContigua + **LoadLista**