COMPLEXIDADE DE ALGORITMOS

É a medida de consumo de recursos, buscando a otimização e a simplicidade durante a execução. A eficiência de desempenho depende: ***tempo gasto e espaço ocupado.***

ANÁLISE DO ALGORITMO (não considera tempo de execução)

- *contagem de operações primitivas*

> *atribuição de valores a variáveis*
> *operações aritméticas*
> *acesso a um arranjo*
> *chamada/retorno de métodos*
> *comparação*

```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      op++;
  return op;
}
```

```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
          op++;
  return op;
}
```

```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<n; ++j)
          op++;
  return op;
}
```

```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; ++i)
      for (int j=0; j<2*i; ++j)
          op++;
  return op;
}
```

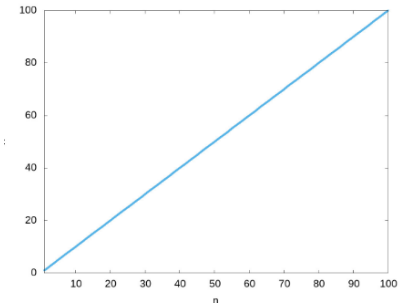
```
int funcao(int n) {
  int op = 0;
  for (int i=0; i<n; i++)
      for (int j=i; j<i+3; j++)
          for (int k=i; k<j; k++)
              op++;
  return op;
}
```

```
int funcao(int n) {
  int op = 1;
  if (n == 1) return op;
  return op + funcao(n-1) + funcao(n-1);
}
```

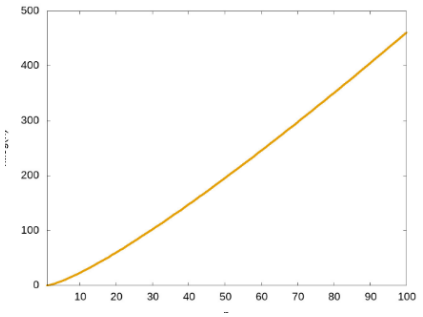
FUNÇÕES (classe de complexidade)

- *constante: ocupa mesma quantidade de recursos*
- *linear: consome recursos de forma diretamente proporcional ao tamanho do problema*
- *NOTAÇÃO O (caracterizar tempo de execução)*
- *limite inferior: menor tempo*
- *limite superior: maior tempo, menor desempenho*
- *NOTAÇÃO ASSINTÓTICA (descrever o comportamento quando n tende ao infinito)*
- ***1 < log n < n < n log n < n2 < n3 < aN***

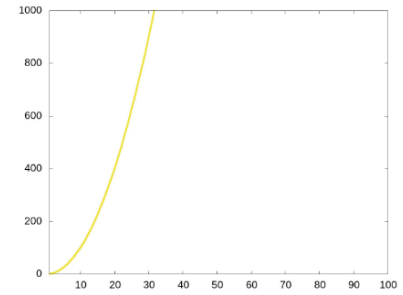
- Linear: n



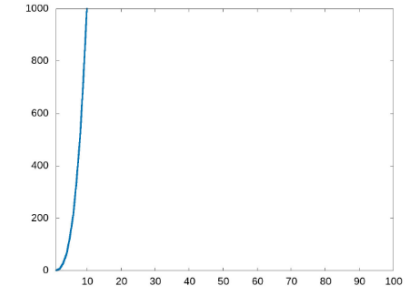
- n-log-n: $n \log n$



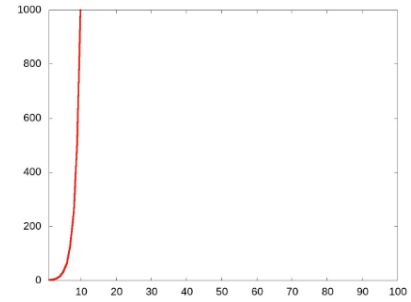
- Quadrática: n^2



- Cúbica: n^3



- Exponencial: a^n



- Exemplos:
  - $5n^2 + 3n \log n + 2n + 5$ é $O(n^2)$
  - $20n^3 + 10n \log n + 5$ é $O(n^3)$
  - $3 \log n + 2$ é $O(\log n)$
  - 2^{n+2} é $O(2^n)$
  - $2n + 100 \log n$ é $O(n)$

RECURSÃO (algoritmo chama a si próprio)

- *fatorial (forma normal)*

$$n! = \prod_{k=1}^{n} k = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1,$$

```
unsigned int fatorial(unsigned int n) {
  unsigned int res = 1;
  for (unsigned int i=2; i<=n; ++i)
      res *= i;
  return res;
}
```

(forma recursiva)

$$n! = \begin{cases} 1, & \text{se } n = 0 \text{ ou } n = 1 \\ n \times (n-1)!, & \text{se } n > 1 \end{cases}$$

```
unsigned int fatorialRec(unsigned int n) {
  if (n <= 1) return 1;
  return n * fatorialRec(n-1);
}
```

- *contagem CRESCENTE (forma normal)*

```
void contagem(int n) {
  for (int i=1; i<=n; i++)
      printf("%d\n", i);
}
```

(forma recursiva)

```
void contagemRec(int n) {
  if (n == 0) return;
  contagemRec(n-1);
  printf("%d\n", n);
}
```

- *contagem DECRESCENTE (forma normal)*

```
void contagemRegressiva(int n) {
  for (int i=n; i>=1; i--)
      printf("%d\n", i);
}
```

(forma recursiva)

```
void contagemRegressivaRec(int n) {
  if (n == 0) return;
  printf("%d\n", n);
  contagemRegressivaRec(n-1);
}
```

ALGORITMOS DE PESQUISA

- ***pesquisa linear** (interativa)*

```
int pesquisa_linear(int *dados, int tam, int valor) {
  for (int i=0; i<tam; i++)
      if (valor == dados[i])
         return i;
  return -1;
}
```

(recursiva)

```
int pesquisaLinearRec(int *dados, int tam, int valor) {
  if (tam == 0) return -1;
  --tam;
  if (dados[tam] == valor) return tam;
  return pesquisaLinearRec(dados,tam,valor);
}
```

- *melhor caso: valor procurado é o primeiro do arranjo*
- *pior caso: o valor procurado NÃO existe*
- *arranjo NÃO precisa estar ordenado*
- *O (n)*
- *Procura um item, comparando-o com cada elemento da coleção, até achar ou chegar no final*

- ***pesquisa binária*** *(interativa)*

```
int pesquisaBinaria(int *dados, int ini, int fim, int valor) {
  while (ini <= fim) {
        int meio = (ini + fim) / 2;
        if (valor == dados[meio])
           return meio;
        else if (valor < dados[meio])
           fim = meio-1;
        else
           ini = meio+1;
  }
  return -1;
}
```

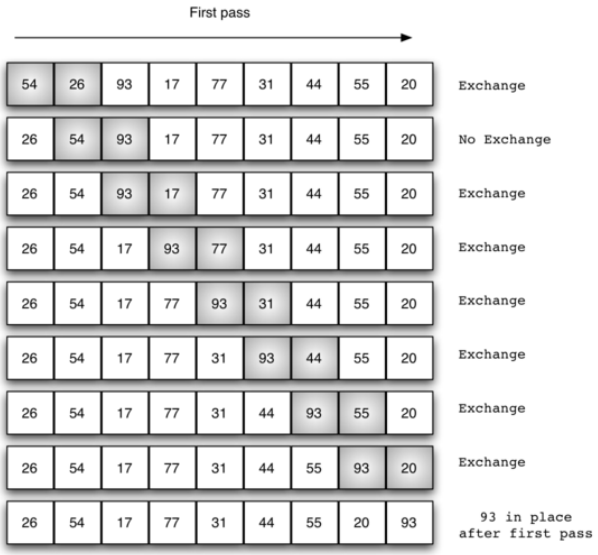
(recursiva)

```
int pesquisaBinariaRec(int *dados, int ini, int fim, int valor) {
  if ( ini > fim ) return -1;
  int meio = (ini + fim) / 2;
  if (valor == dados[meio])
     return meio;
  else if (valor < dados[meio])
     return pesquisaBinariaRec(dados, ini, meio-1, valor);
  else
     return pesquisaBinariaRec(dados, meio+1, fim, valor);
}
```

- *somente para coleções ordenadas*
- *O (log n)*
- *Estratégia básica:*
  - ➔ *Verifica o elemento central: se encontrou, a busca termina*
  - ➔ *Se o item for menor que o central, considera apenas a parte abaixo do elemento central*
  - ➔ *Se o item for maior que o central, considera apenas a parte acima do elemento central*
- *Trabalha subdividindo a coleção e aplicando sempre a estratégia básica*

ALGORITMOS DE ORDENAÇÃO

- ***bubble sort***



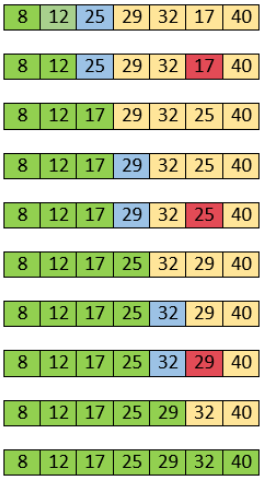

```
void bubbleSort(int *dados, int tam) {
  int trocou;
  do {
     trocou = 0;
     --tam;
     for (int i=0; i<tam; ++i) {
         if (dados[i] > dados[i+1]) {
            int aux = dados[i];
            dados[i] = dados[i+1];
            dados[i+1] = aux;
            trocou = 1;
         }
     }
  } while (trocou);
}
```

- *Complexidade: O(n) (melhor caso) ou O(n2) (pior caso)*
- *'dados [i]' vai analisar o número da esquerda, e 'dados [i + 1]' analisa o número da direita*
- *'aux' serve para armazenar o valor que será trocado de posição, assim não se perde o valor*

exemplo: 26 se transforme no 54, sem perder o valor 54

- ***dados[i] = 54** é maior que **dados [i + 1] = 26***
- *se for maior, 'aux' recebe o valor atribuído a dados [i], **aux = 54***
- *agora dados [i] recebe o valor de dados [i = 1], **dados [i] = 26***
- *dados [i + 1] deve receber o valor de aux, **dados [i + 1] = 54***

- ***selection sort***




```
void selectionSort(int *dados, int tam) {
  for (int i=0; i<tam-1; ++i) {
      int men = i;
      for (int j=i+1; j<tam; ++j)
          if ( dados[j] < dados[men] ) men = j;
      if ( men != i ) {
         int aux = dados[men];
         dados[men] = dados[i];
         dados[i] = aux;
      }
  }
}
```

- *Complexidade: O(n2) (melhor e pior caso)*
- *Quanto maior a lista de números, + espaço na memória ocupado, + tempo de execução*

exemplo: *procura o menor valor da lista e põe como primeiro elemento,* ***men = i = 8***
enquanto isso, ***j = i + 1*** *percorre o espaço sobrando*
apenas preenche com o 12 por ser maior
8 / 12
entretanto, a partir da 2 linha, o menor valor passa a ser o 17
*(****dados [j] = 12*** *menor* ***dados [men] = 25****)*
logo, ***men = j = 17***
como men será diferente de i, é efetuado a troca de elementos
aux = dados [men] = 25
dados [men] = 25 *é* ***dados [i] = 17***
dados [i] = 17 *é* ***aux = 25***
8 / 12 / 17 / 25

- ***insertion sort***

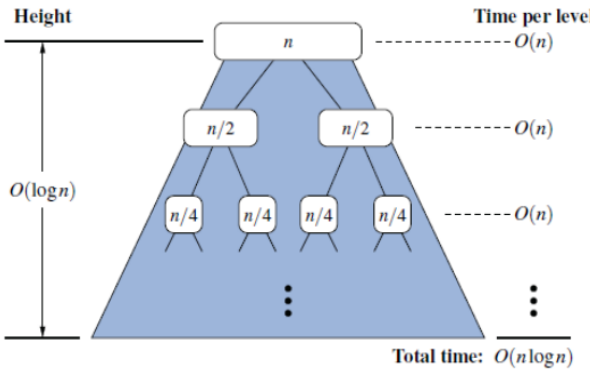
0	1	2	3	4	5	6	7	8	9
5	7	8	1	10	9	4	6	3	2
5	7	8	1	10	9	4	6	3	2
5	7	8	1	10	9	4	6	3	2
1	5	7	8	10	9	4	6	3	2
1	5	7	8	10	9	4	6	3	2
1	5	7	8	9	10	4	6	3	2
1	4	5	7	8	9	10	6	3	2
1	4	5	6	7	8	9	10	3	2
1	3	4	5	6	7	8	9	10	2
1	2	3	4	5	6	7	8	9	10

```
void insertionSort(int *dados, int tam) {
  for (int i=1; i<tam; ++i) {
       int base = dados[i];
       int j = i-1;
       while ( j>=0 && base < dados[j] ) {
             dados[j+1] = dados[j];
             --j;
       }
       dados[j+1] = base;
  }
}
```

- *Complexidade: O(n) (melhor caso) ou O(n2) (pior caso)*

exemplo: ***base = dados [i] = 7****, elemento a frente*
(j = i - 1) = 5*, elemento anterior*
ordem permanece pois 5 é menor que 7
5 / 7
base = dados [i] = 8*, elemento da frente*
(j = i - 1) = 7*, elemento anterior*
dados [j + 1] = dados [j] = 7
base passa a ser o valor 7
ordem muda, já que a base é menor dados [j]
5 / 7 / 8

- ***merge sort***

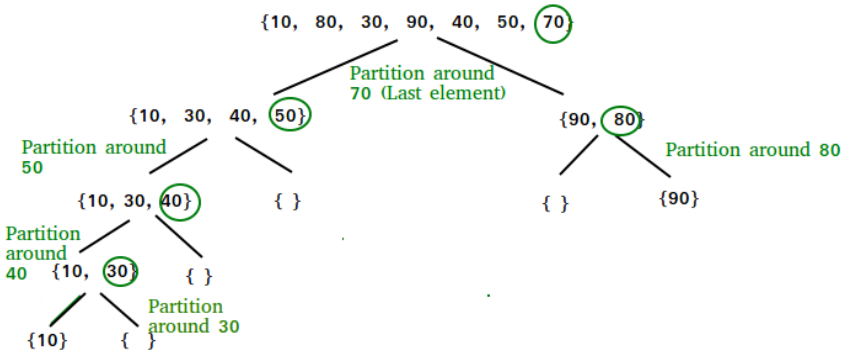



```
void merge(int *dados, int ini, int meio, int fim) {
  int p = ini, q = meio+1, k=0;
  int *aux = new int[fim-ini+1];
  for (int i = ini; i <= fim; i++){
      if (p > meio)                  aux[k++] = dados[q++];
      else if (q > fim)              aux[k++] = dados[p++];
      else if( dados[p] < dados[q]) aux[k++] = dados[p++];
      else                           aux[k++] = dados[q++];
  }
  for (int p=0; p<k; p++) dados[ini++] = aux[p];
  delete[] aux;
}

void mergeSort(int *dados, int ini, int fim) {
  if ( ini >= fim ) return;
  int meio = (ini + fim) / 2;
  mergeSort(dados, ini, meio);
  mergeSort(dados, meio+1, fim);
  merge(dados,ini,meio,fim);
}
```

- *Altura da árvore é log n*
- *Tempo gasto em cada nível: O(n)*
- *Tempo de execução: O(n log n)*
- *Consiste de 3 etapas*
  - ***Divisão**: se há algo a ordenar, divide os dados de entrada em duas (ou mais) partes e executa o algoritmo sobre cada uma das partes; se não há nada a ordenar, retorna a solução*
  - ***Conquista**: cada parte dos dados é classificada recursivamente*
  - ***Combinação**: quando cada subconjunto está classificado (internamente), eles devem ser combinados (merge) realizando-se uma intercalação*
- *Permite implementação recursiva*

- ***quick sort***



- *Melhor caso: O (n log n) / pior caso: O (n2)*
- *escolher **pivô***
- *esquerda: menores elementos / direita: maiores elementos*

- *hoare(pivô: elemento central)*

```
void quickSort(int *dados, int ini, int fim) {
  int i = ini, j = fim, pivo = dados[(ini+fim)/2];
  while (i <= j) {
        while (dados[i] < pivo)
              ++i;
        while (dados[j] > pivo)
              --j;
        if (i <= j) {
           int aux = dados[i];
           dados[i] = dados[j];
           dados[j] = aux;
           ++i;
           --j;
        }
   }
   if (ini < j) quickSort(dados,ini,j);
   if (i < fim) quickSort(dados,i,fim);
}
```

- *lomuto (pivô: primeiro ou último elemento)*

```
int particiona(int *dados, int ini, int fim) {
  int pivo = dados[fim];
  int i = ini-1;
  for (int j=ini; j<fim; ++j) {
      if (dados[j] < pivo) {
         ++i;
         int aux = dados[i]; dados[i] = dados[j]; dados[j] = aux;
      }
  }
  if (pivo < dados[i+1]) {
     int aux = dados[i+1]; dados[i+1] = dados[fim]; dados[fim] = aux;
  }
  return i+1;
}

void quickSort(int *dados, int ini, int fim) {
  if (ini < fim) {
     int pivo = particiona(dados, ini, fim);
     quickSort(dados, ini, pivo-1);
     quickSort(dados, pivo+1, fim);
  }
}
```

COMPARAÇÃO

- *estáveis: bubble sort, insertion sort, merge sort*
- *instáveis: selection sort, quick sort*
- *mais rápido com a lista ordenada: bubble sort*
- *mais rápido com a lista invertida: quick sort (hoare)*
- *mais rápido com a lista aleatória: quick sort (lomuto)*

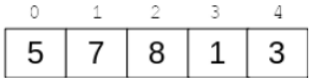
ARMAZENAMENTO

- *versões recursivas necessitam de memória da pilha*
- *versões in-place utilizam apenas o espaço da própria coleção, NÃO necessitam de memórias auxiliares (quick sort, merge sort)*

ESTRUTURAS LINEARES

- *cada elemento da estrutura é um nó (nodo)*
- *nullptr (estrutura vazia)*
- *exemplo: (2 + 3) * 4 == pré- fixada (ordem; * + 2 3 4) / pós- fixada (ordem 2 3 + 4 *)*

➔ *sequencial*

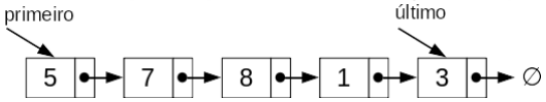


- *Implementação é feita com vetores (arranjos ou arrays), que podem ser alocados de forma estática ou dinâmica*

➔ *encadeada (não é possível prever o num. de entradas)*

- *Implementada através de uma ligação (referência ou armazenamento de endereço) entre os nodos*

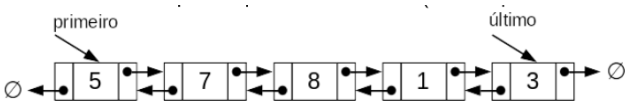
➢ *simplesmente encadeada*




```
struct Node {
  char info;
  Nodo *next;
  Nodo(char l) {
    info = l;
    next = nullptr;
  }
};
```

- *cada nodo pode armazenar uma referência para o próximo elemento*

➢ *duplamente encadeada*




```
struct Node {
  int info;
  Node *prev, *next;
  Node(int i) {  info = i;  prev = next = nullptr;  }
};
```

- *cada nodo pode armazenar referência para o elemento anterior e para o próximo elemento*

- ***LISTA (sequência)***

- *Inserção (no início, no m ou em uma posição específica)*
- *Remoção (no início, no m ou em uma posição específica)*
- *Busca e acesso (através de índice ou através da informação de algum campo)*
- *Alteração*

➔ ***arranjo (maior desempenho)***

- Inserção no início

```
Node *node = new Node(info);
node->next = head;
head = node;
if ( tail == nullptr ) tail = node;
```

- Inserção no meio

```
// aux aponta para nodo antes do qu
// ant aponta para nodo anterior a
Node *node = new Node(info);
ant->next = node;
node->next = aux;
```

- Inserção no final

```
Node *node = new Node(info);
tail->next = node;
tail = node;
```

- Remoção do início

```
if ( head != nullptr ) {
   Node *aux = head;
   head = head->next;
   if ( head == nullptr ) tail = nullptr;
   delete aux;
}
```

- Remoção do meio

```
// aux aponta para nodo que se quer remov
// ant aponta para nodo anterior a aux
ant->next = aux->next;
if ( aux->next == nullptr ) tail = ant;
delete aux;
```

- Remoção do final

```
// ant aponta para nodo anterior a tail
ant->next = nullptr;
delete tail;
tail = ant;
```

- ***bool add****: insere o elemento no final da lista*
- ***bool add (index)****: insere o elemento em um índice específico da lista*
- ***bool get (index)****: retorna o elemento do índice especificado (por referência)*
- ***bool set (index)****: atribui o elemento para a posição do índice especificado*
- ***bool remove (index)****: remove o elemento do índice especificado da lista*

➔ ***encadeada (menor desempenho)***

- ***bool push_front****: insere o elemento no início da lista*
- ***bool push_back****: insere o elemento no final da lista*
- ***bool insert (index)****: insere o elemento no índice especificado*
- ***bool pop_front****: remove o elemento do início da lista*
- ***bool pop_back****: remove o elemento do final da lista*
- ***bool remove (index)****: remove o elemento do índice especificado da lista*
- ***bool get (index)*** *e* ***bool set(index)****: igual ao arranjo*
- ***bool contains****: verificar se o elemento existe na lista*
- ***int indexof****: retorna o índice da primeira ocorrência do elemento na lista*
- ***int indexof(pos)****: retorna o índice da próxima ocorrência do elemento a partir da posição especificada*

- Inserção no início

```
Node *node = new Node(info);
if ( head == nullptr ) { head = tail = node; }
else {
  node->next = head;
  head->prev = node;
  head = node;
}
```

- Inserção no meio

```
// aux aponta para nodo antes do qual se quer
Node *node = new Node(info);
node->prev = aux->prev;
node->next = aux;
(aux->prev)->next = node;
aux->prev = node;
```

- Inserção no final

```
Node *node = new Node(info);
node->prev = tail;
tail->next = node;
tail = node;
```

- Remoção do início

```
if ( head != nullptr ) {
   Node *aux = head;
   head = head->next;
   if ( head == nullptr ) tail = nullptr;
   else head->prev = nullptr;
   delete aux;
}
```

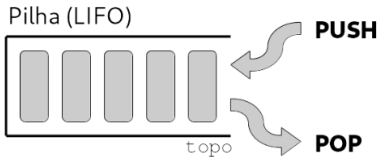
- Remoção do meio

```
// aux aponta para nodo que se quer remover
(aux->prev)->next = aux->next;
if ( aux->next == nullptr ) tail = aux->prev;
else (aux->next)->prev = aux->prev;
delete aux;
```

- Remoção do final

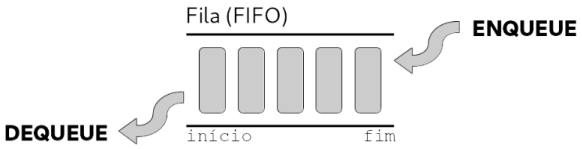
```
Node *aux = tail;
tail = tail->prev;
if ( tail == nullptr )  head = nullptr;
else tail->next = nullptr;
delete aux;
```

- ***PILHA (Last in, First out)***



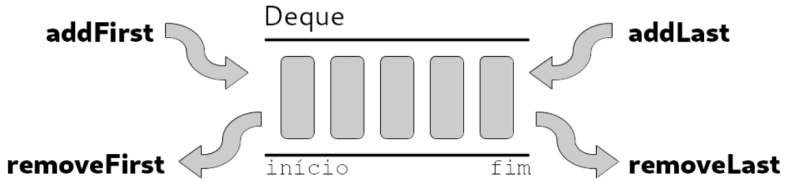
- ***bool push**: insere elemento no topo da pilha*
- ***bool pop**: remove e retorna o elemento do topo da pilha (referência)*
- ***bool top**: retorna o elemento do topo da lista (referência)*

- ***FILA (First in, First out)***



- ***bool enqueue**: insere o elemento no final da fila*
- ***bool dequeue**: remove e retorna o elemento do início da fila*
- ***bool head**: retorna o elemento do início da fila*
- *verifica o **resto (%)***

- ***DEQUE (Entram e Saem por qualquer extremidade)***



- ***bool addFirst****: insere o elemento no início do deque*
- ***bool addLast****: insere o elemento no fim do deque*
- ***bool removeFirst****: remove e retorna o elemento do início do deque*
- ***bool removeLast****: remove e retorna o elemento do fim do deque*
- ***bool head****: retorna o elemento do início da lista*
- ***bool tail****: retorna elemento do fim do deque*