Listas encadeadas são utilizadas para armazenar uma coleção de elementos. É uma alternativa a vetores

Listas:	Vetores
Não contiguo	Contiguo na memória
Acesso: $O(n)$	Acesso: $O(1)$
Inserção: $O(1)$	Inserção: $O(n)$

Lista prioriza a manipulação e não o acesso

Ex:
```
int maior (int v[], int n){
1   int m = v[0]
n-1 for (int i=1; i<n; i++)
2(n-1)   if(v[i]>m) m = v[i]
1   return m;}
```

Logo é $O(n)$

* Dizer que um algoritmo consome tempo $O(f(n))$, onde f é uma função e n é algo que caracteriza a entrada, significa dizer que o algoritmo consome, no máximo,

$$c\,f(n)$$

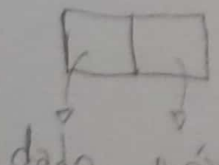
operações para $\forall\ n \geq n_0$, onde $c > 0$ e $n_0 \geq 0$.

Funções

$f(n) = 1$ Constante
$f(n) = \lg n$ logaritmico
$f(n) = n$ linear
$f(n) = n \lg n$ linearítmica
$f(n) = n^k$, $k \geq 2$ polinomial
$f(n) = k^n$, $k \geq 2$ exponencial

Implementação de listas encadeadas

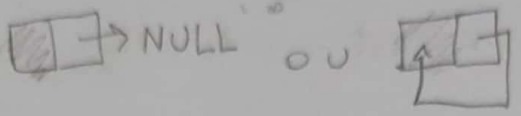
• Nó



Em código

```
typedef struct no{
    int dado;
    struct no *próx;
}nó;
```

Lista simplesmente encadeada

Ex: lista vazia 

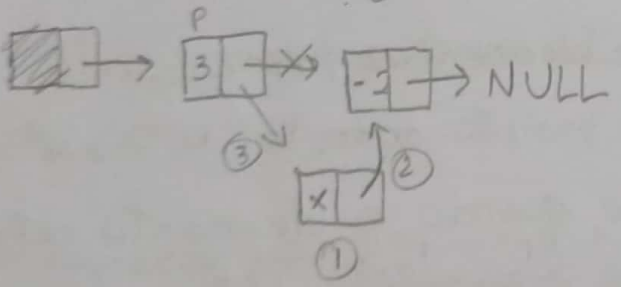
é o nó cabeça

- Criação da lista

```
no *le = malloc(sizeof(no));
le->prox = NULL
```

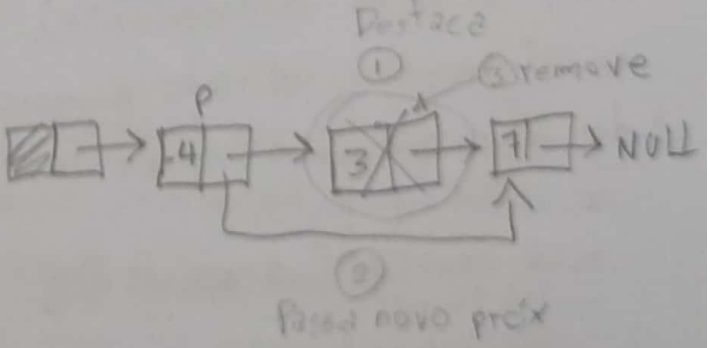
- Inserção

```
void insere (no *p, int x){
    no *novo = malloc (sizeof(no));
    novo->dado = x;
    novo->prox = p->prox;
    p->prox = novo; }
```



- Remoção

```
int remove (no *p){
    no *lixo = p->prox
    if (lixo != NULL){
        int x = lixo->dado;
        p->prox = lixo->prox;
        free (lixo)};
        return x; }
```



Ordenação

- $O(n^2)$ (quadrático)
  - Inserção
  - Bolha
  - Seleção
- $O(n)$ (linear)
  - shell
  - contagem
  - distribuição
- $O(n \lg n)$ (linearitmico)
  - quicksort
  - mergesort
  - heapsort

Ordenação por inserção

Dado um vetor $v[0...n-1]$ com n elementos, para cada i de $0...n-1$, insere $v[i]$ na posição correta do subvetor $v[0...i]$. O pior caso ocorre quando o elemento retirado é comparado com todos à esquerda (vetor ordenado em ordem decrescente). Já o melhor caso ocorre quando o elemento retirado é maior que o antecessor (vetor ordenado em ordem crescente) $O(n)$. Caso médio: $O(n^2)$

```
void insercao (int *V, int n) {
→ int elem, i;
→ for(i=1, i<n, i++){
→→ elem = V[i];
→→ for(j=i-1, j≥0 and V[j]>elem){
→→→ V[j+1] = V[j];}
→→ V[j+1] = elem; }
}
```

Ordenação por seleção

Para cada i, de 0...n-1, selecione o menor elemento do vetor v[i...n-1] e insira-o em v[i].

- Custo de encontrar o menor: $O(n)$
- Pior caso $n+(n-1)+(n-2)+...+2 = O(n^2)$

Estabilidade: Dizemos que um algoritmo de ordenação é estável se ele conserva a ordem relativa de elementos iguais. Diferentemento do algoritmo de ordenação por inserção, o algoritmo de seleção é instável

Valgrid

1) Compila com debugação ativada
2) Roda com valgrind

3 5 6 8 2 1

Ordenação por intercalação (MergeSort)

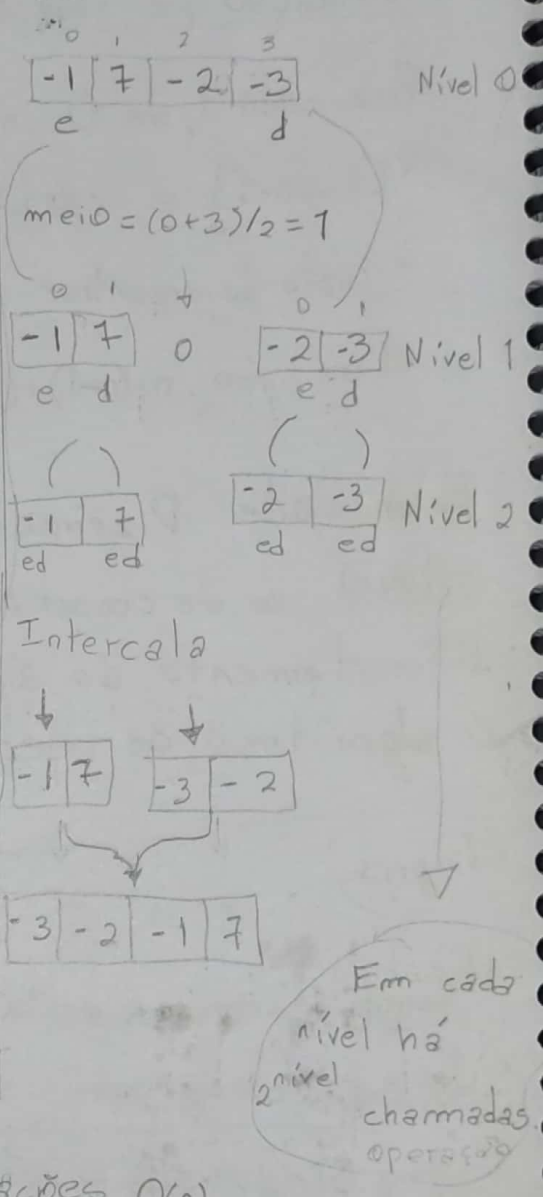
- Divisão e conquista

Dado um vetor, divide-o no meio e ordena recursivamente cada metade Na volta, intercala as duas metades e obtém o vetor ordenada. Complexidade $O(n \lg n)$ linearítmico

```
void mergesort (int *v, int e, int d){
    if (d<=e) return;
    int meio = (e+d)/2;
    mergesort (v, e, meio);
    mergesort (v, meio+1, d);
    intercala (v, e, meio, d);
}

void intercala (int *v, int e, int meio, int d){
    int *v2 = malloc ((d-e+1)*sizeof(int))
    int i=e, j=meio+1, k=0
    while (i<=meio && j<=d) {
      if (v[i]<=v[j]) v2[k++]=v[i++];
      else v2[k++]=v[j++];}
    while (i<=meio) v2[k++]=v[i++]
    while (j<=d) v2[k++]=v[i++]
    for (k=0, i=e; i<=d; k++, i++) v[i]=v2[k];
}
```

Ex:



1) A cada nível da árvore, fazemos operações $O(n)$

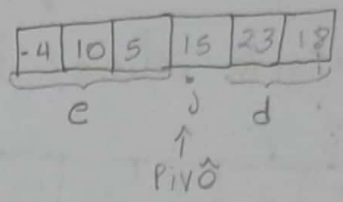
2) Logo o total de operações é a quantidade de níveis da árvore multiplicado por $O(n)$.

3) Quantos níveis possui uma árvore gerada para um vetor de n elementos, $n = 2^k$? É $\lg n$

Logo, como cada nível custa $O(n)$, a complexidade total é $O(n \lg n)$.

Quicksort

Ex:

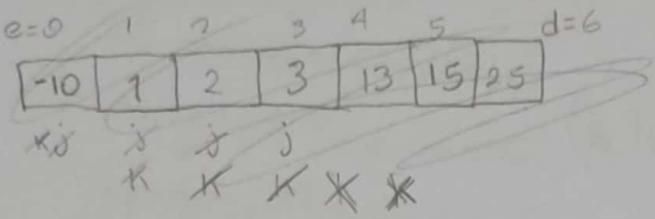



```
void quicksort(int *V, int e, int d)
{  if (d<=e) return;
   int particiona (v,e,d)
   quicksort (v,e, j-1)
   quicksort(v,j+1,d)

}

int particiona (int*V,int e, int d)
   int tam = d-e+1
   int * menores = malloc(sizeof(int)*tam)
   int *maiores=malloc(sizeof(int)* tam)
   int pivo= V[d]
   int imenores=0,imaiores=0;
   for (int k=e; k<d;k++){
     if(V[k]<pivo)
        menores[imenores++]=V[k];
     else
        maiores[maiores++]=V[k];}
  int i=0;
  for(k=0;k<imenores;k++)
    V[i++]=menores[k];
  V[i++]=pivo;
  for(k=0;k<imaiores;k++){
     V[i++]=maiores[k]
  free(maiores);
  free(menores);
  return posPivo}
```

Implementação inline do particiona




```
int particiona (int *v, int e, int d){
    int pivo = v[d];
    int j = e;
    for (int k = e; k < d; k++){
        if (v[k] <= pivo){
            troca(v, k, j)
            j++;}
    troca (v, j, d)
    return j
}
```

```
void troca (int *v, int i, int j)
{ int tmp = v[i];
  v[i] = v[j]
  v[j] = tmp
}
```

Caso médio
$O(n \lg n)$

Para resolver o pior caso do quicksort $O(n^2)$

- Número aleatório
  - Pseudo aleatoriedade
  - Problema para definir a semente
- mediana de 3 elementos
  - garante que o pivô não é o menor e nem o maior

Como implementar mediana de 3?

① V[e], V[d] e V[(e+d)/2]

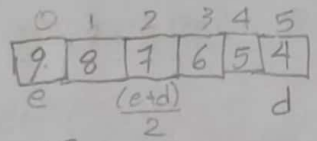
② Como calcular?

if (V[(e+d)/2] > V[d]) troca (V, d, (e+2)/2)

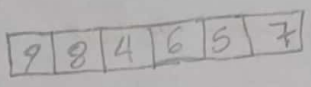
if (V[(e+d)/2 < V[e]) troca (V, (e+d)/2, e)

if (V[(e+d)/2 < V[d]) troca (V, (e+d)/2, d)

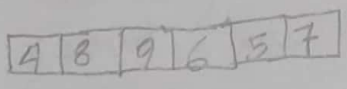
Ex:



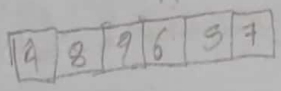
① 4<7? => Sim



② 4<9? => Sim



③ 9<7? => Não



Colocando as 3 instruções acima numa rotina mediana3 (V, e, d)

```
void quicksort(int *V, int e, int d)
{ if (d <= return)
    mediana3(V,e,d);
    j = particiona (V,e,d);
    quicksort (V, e, j-1);
    quicksort (V, j+1, d);
```

Quando compensa ordenar para fazer buscas?

Ordenar + Buscar = $O(n \lg n + \lg n)$

$O(n \lg n)$ $O(\lg n)$ $= O((n+1) \lg n)$

X

Busca sequencial $= O(n)$

Portanto, para uma única busca, não compensa ordenar.

Por outro lado, se a quantidade de buscar for grande.
(isto é, muito mais que n buscas)
Compensa pois

Ordenar + n buscas = $O(n \lg n)$

$O(n \lg n)$ $n(O \lg n)$

Por outro lado, n buscas sequenciais = $O(n^2)$

$n \cdot O(n)$

Consideremos o problema dos números proibidos

↳ Temos uma lista com n números proibidos ($n \leq 14000$) dada na entrada

↳ Cada número varia entre 1 e 2^{31}

↳ Para cada consulta, desejo classificar o número se é proibido ou não.

Sol.1: Salvo os números proibidos num vetor de tamanho n e para cada consulta, faço uma busca sequencial

↳ Custo elevado se forem muitas buscas: $O(m \cdot n)$

Sol.2: Ordenar e buscar: boom se tiver muitas buscas

$O(n \lg n) + O(m \lg n) = O((n+m) \lg n)$

Sol.3: $O(m+n)$

↳ Declara um vetor v com 2^{31} elementos e zera o vet

↳ Para cada número proibido i, define $v[i]=1$

↳ Para cada busca com número j, se $v[j]=0$, então o número não é proibido, se $v[j]=1$, o número j é proibido $O(m)$

Custo: $O(1)+O(n)+O(m)=O(n+m)$

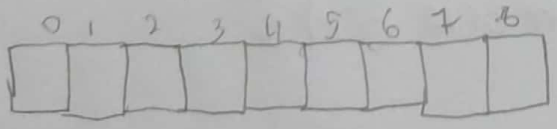
Gasta muita memória: 4×2^{31} Bytes $= 4 \times 2 \times 2^{30} =$ 8GB

```
int hash (int x){
    return x%140000;
}

int main(){
  int N;
  int i, num;
  int *V=calloc(sizeof(int), 140000);
  scanf("%d", &N);
  for(i=0; i<N; i++){
     scanf("%d", &num);
     V[hash(num)]=num;
  }
  while(scanf("%d", &num)!=EOF){
     int h = hash(num);
     if(V[h]==num) printf("Proibido\n");
     else if(V[h]==0) printf("Não é proibido\n");
     else if(V[h]!=num) printf("Colisão");
  }//while
  return 0;}
```

Métodos para resolver questões

① Endereço aberto



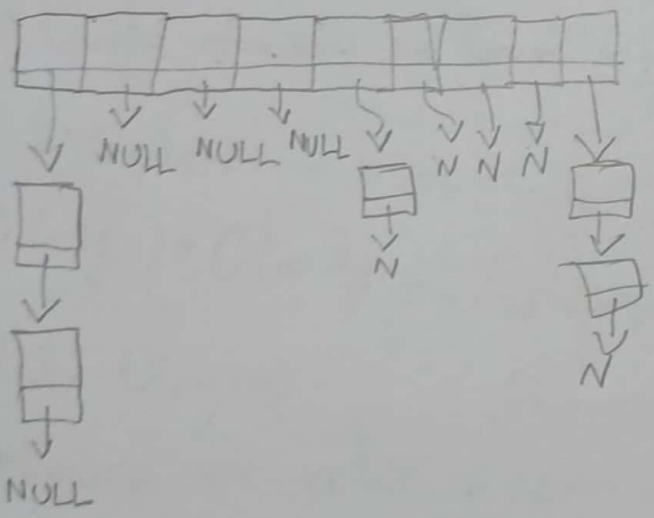
Critério de parada é um local vazio

A função hash estabelece o início p/ fazer a busca no vetor, para-se quando encontrar vazio. No caso de colisões, salva-se na próxima posição disponível

Problemas: A tabela hash não pode ficar cheia. Quando estiver por encher, é necessário redimensioná-la a um custo ①(n) (em que n é o tamanho da hash)

② A busca numa tabela hash <u>densa</u> é pior. Quanto menor a esparsidade (ou maior a densidade), pior a busca

② Encadeamento separado



Como listas não ocupam posições contiguas na memória, a busca pode ser lenta por conta de muitas falhas que podem ocorrer na memória cache