

```

int main(int argc, char** argv)
{
    char c = 0;
    char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
    int speed = 2; // sets max moves per row
    int moves_to_go = 2;
    int full = 0; // whether board is full
    init(); // initialize board an tetrominoes

```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Alocação sequencial

```

// process user action
c = getchar(); // get new action
if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
if (c == commands[3]) moves_to_go=0;

// scroll down
if (!moves_to_go--)
{
    if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
    {
        cramp_tetromino();
        remove_complete_lines();
        cur = &tetrominoes[rand() % NUM_POSES];
        state[0] = (WIDTH - cur->width)/2;

```

Vetores (arranjos, *arrays*)

```
tipo nome[capacidade];
```

```
tipo nome[] = {e1, ..., eN};
```

Aloca sequência contígua de variáveis do mesmo tipo

- ▶ capacidade é **constante** do tipo unsigned int
- ▶ nome é **ponteiro** para primeira variável da sequência

u[0]	u[1]	u[2]	u[3]
1	2	3	4

v[0]	v[1]	v[2]	v[3]	v[4]
2.0	4.0	6.0	8.0	-1.0

```
1 int i;  
2 int u[] = {1, 2, 3, 4}; /* vetor de inteiros */  
3 double v[5]; /* vetor de doubles */  
4 for (i = 0; i < 4; i++) v[i] = 2*u[i];  
5 v[i] = -1.0;
```

Vetores: Acesso e definição

```
tipo nome[capacidade];
```

- ▶ Acesso: `(tipo) nome[indice]`
- ▶ Definição: `nome[indice] = (tipo) valor;`
- ▶ **Cuidado:** Não há verificação de validade no acesso/definição

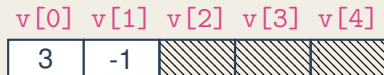
```
1  /* Código compila e talvez rode */
2  int u[] = {1,2,3,4};
3  int v[5];
4  int i;
5  for (i = 0; i <= 5; i++) {
6      /* indefinido para i > 3 */
7      v[i] = 2*u[i];
8      printf("%d ", v[i]);
9  }
```

Vetores: Tamanho efetivo

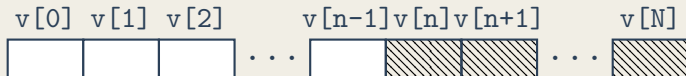
```
tipo nome[capacidade];
```

- ▶ **Tamanho efetivo**: número de elementos definidos
- ▶ **Não** deve ultrapassar capacidade

```
1  int v[5]; /* capacidade = 5 */
2  int n = 0; /* tam. efetivo */
3  v[0] = 3; v[1] = -1;
4  n = 2; /* tam. efetivo = 2 */
```



Vetores: Cheio e Vazio



► **Vazio:** `n=0`

► **Cheio:** `n=N`

```
1 #define N 10
2 int v[N];
3 int n;
4 v[0] = 32;
5 v[1] = 10;
6 n = 2;
```

Vetores: Capacidade inferida

`sizeof(v)/sizeof(v[0])`

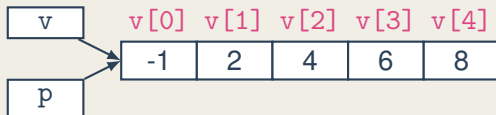
Capacidade de vetor definido no mesmo escopo

```
1 void capacidade(int u[]) {  
2     static int v[5];  
3     int w[] = {1,2,3};  
4     printf("%lu", sizeof(v)/sizeof(v[0])); /* 5 */  
5     printf("%lu", sizeof(w)/sizeof(w[0])); /* 3 */  
6     printf("%lu", sizeof(u)/sizeof(u[0])); /* indefinido */  
7 }
```

Vetores e ponteiros

Vetores são representados por um ponteiro para o primeiro elemento

```
1 int i;  
2 int v[5];  int *p;  
3 p = v;  
4 for (i = 0; i < 5; i++) v[i]=2*i;  
5 *p = -1;
```



Funções de vetores

tipo nome (int v[], int n, ...) bloco

argumentos: ponteiro para vetor e tamanho (efetivo) do vetor

```
1 int maximo (int v[], int n) {  
2     int i, maior = *v;  
3     for (i=1;i<n;i++) if (v[i] > maior) maior = v[i];  
4     return maior;  
5 }  
6  
7 int maximo2 (int *v, int n) {  
8     int i, maior = v[0];  
9     for (i=1;i<n;i++) if ( *(v+i) > maior ) maior = *(v+i);  
10    return maior;  
11 }
```


Objetivo: Determinar se dado valor ocorre em um certo vetor.

Busca linear ou sequencial

```
1 /* Recebe inteiro x e vetor v[0..n-1]
2    e retorna k tal que v[k] = x ou k = n
3    se o vetor não contém x */
4 int Busca (int x, int v[], int n) {
5     int k;
6     for (k = 0; k < n; k++)
7         if (v[k] == x) return k;
8     return n;
9 }
```

Busca linear ou sequencial

Versão sem if:

```
1  /* Recebe inteiro x e vetor v[0..n-1]
2     e retorna k tal que v[k] = x ou k = n
3     se o vetor não contém x */
4  int Busca (int x, int v[], int n) {
5      int k;
6      for (k = 0; k < n && v[k] != x; k++);
7      return k;
8  }
```

Busca linear ou sequencial

Versão com `while` e sem `if`:

```
1  /* Recebe inteiro x e vetor v[0..n-1]
2     e retorna k tal que v[k] = x ou k = n
3     se o vetor não contém x */
4  int Busca (int x, int v[], int n) {
5     int k = 0;
6     while (k < n && v[k] != x) k++;
7     return k;
8 }
```

Qual o problema?

```
1  /* Recebe inteiro x e vetor v[0..n-1]
2     e retorna k tal que v[k] = x ou k = n
3     se o vetor nao contem x */
4  int Busca (int x, int v[], int n) {
5      int k = 0;
6      while (v[k] != x && k < n) k++;
7      return k;
8  }
9
```

Qual o problema?

```
1  /* Recebe inteiro x e vetor v[0..n-1]
2     e retorna k tal que v[k] = x ou k = n
3     se o vetor nao contem x */
4  int Busca (int x, int v[], int n) {
5     int k = 0;
6     while (v[k] != x && k < n) k++;
7     return k;
8  }
9
```

Código acessa $v[n]$ quando x não está no vetor.

Deselegante (uso desnecessário de indicador de passagem):

```
1 /* Recebe inteiro x e vetor v[0..n-1]
2    e retorna k tal que v[k] = x ou k = n
3    se o vetor nao contem x */
4 int Busca (int x, int v[], int n) {
5     int k = 0;
6     int achou = 0
7     while (k < n && !achou) {
8         if (v[k] == x) achou = 1;
9         else k++;
10    }
11    return k;
12 }
```

Ainda mais deselegante (sem término prematuro):

```
1 /* Recebe inteiro x e vetor v[0..n-1]
2    e retorna k tal que v[k] = x ou k = n
3    se o vetor nao contem x */
4 int Busca (int x, int v[], int n) {
5     int i, k = 0;
6     int achou = 0
7     while (k < n) {
8         if (v[k] == x) { achou = 1; i = k; }
9         k++;
10    }
11    if (achou) return i;
12    else return n;
13 }
```

Objetivo: Fazer apenas uma comparação a cada iteração do laço de iteração da busca.

Suponha que $n < \text{capacidade}$

```
1 /* Recebe inteiro x e vetor v[0..n-1]
2    e retorna k tal que v[k] = x ou k = n
3    se o vetor nao contem x */
4 int Busca (int x, int v[], int n) {
5     int k = 0;
6     /* faz duas comparações por iteração */
7     while (k < n && v[k] != x) k++;
8     return k;
9 }
```


Objetivo: Fazer apenas uma comparação a cada iteração do laço de iteração da busca

Suponha que $n < \text{capacidade}$

Busca linear com sentinela

```
1 /* Recebe inteiro x e vetor v[0..n-1]
2    e retorna k tal que v[k] = x ou k = n
3    se o vetor nao contem x */
4 int Busca (int x, int v[], int n) {
5     int k = 0;
6     v[n] = x; /* sentinela - garante término */
7     while (v[k] != x) k++; /* uma comparação por iteração */
8     return k;
9 }
```

Sentinela

valor especial usado para indicar uma condição de término

- ▶ `'\0'` é sentinela para strings (veremos adiante)
- ▶ `EOF` é sentinela para arquivos (veremos adiante)
- ▶ `-1` é possível sentinela para um vetor de inteiros positivos

Sentinelas são úteis evitar comparações

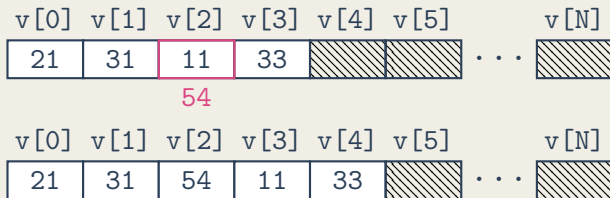
```
1 int Busca (int x, int v[], int n) {  
2     int k = 0;  
3     v[n] = x;  
4     while (v[k] != x) k++;  
5     return k;  
6 }
```

Inserção em vetor

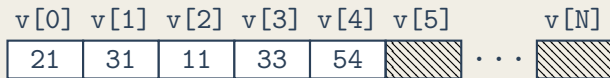
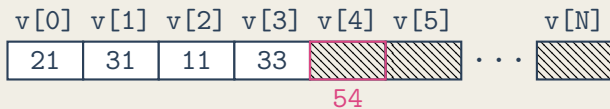
Dados um vetor $v[0..n-1]$, um inteiro $0 \leq k \leq n$ e um valor x , inserir x entre as posições k e $k+1$

- ▶ Capacidade $> n$
- ▶ Manter ordem relativa
- ▶ Novo vetor: $v[0..n]$

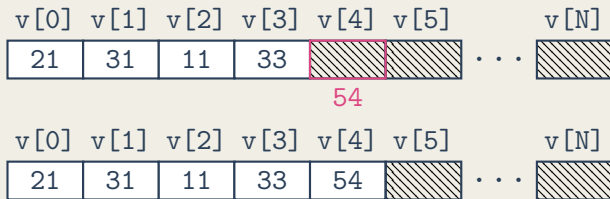
`insere(v, 4, 54, 2);`



Mais simples: Inserção em vetor no final ($k=n$)



Mais simples: Inserção em vetor no final ($k=n$)



```
1 int insere_no_fim(int v[], int n, int x) {  
2     v[n++] = x; /* v[n] = x; n++; */  
3     return n; /* devolve novo tamanho */  
4 }
```

Caso geral: Inserção em vetor

inserção em posição arbitrária com preservação da ordem

`insere(v, 4, 54, 2);`



Caso geral: Inserção em vetor

inserção em posição arbitrária com preservação da ordem

`insere(v, 4, 54, 2);`



```
1 int insere(int v[], int n, int x, int k) {  
2     int j;  
3     for (j = n; j > k; j--) v[j] = v[j-1];  
4     v[k] = x;  
5     return n + 1; /* devolve novo tamanho */  
6 }
```

Inserção em vetor

Inserção em posição arbitrária

```
1 int insere(int v[], int n, int x, int k) {  
2     int j;  
3     for (j = n; j > k; j--) v[j] = v[j-1];  
4     v[k] = x;  
5     return n + 1;  
6 }
```

Custo computacional:

► $k=0$?

Inserção em posição arbitrária

```
1 int insere(int v[], int n, int x, int k) {  
2     int j;  
3     for (j = n; j > k; j--) v[j] = v[j-1];  
4     v[k] = x;  
5     return n + 1;  
6 }
```

Custo computacional:

- ▶ $k=0$? n deslocamentos
- ▶ $k=n$?

Inserção em posição arbitrária

```
1 int insere(int v[], int n, int x, int k) {  
2     int j;  
3     for (j = n; j > k; j--) v[j] = v[j-1];  
4     v[k] = x;  
5     return n + 1;  
6 }
```

Custo computacional:

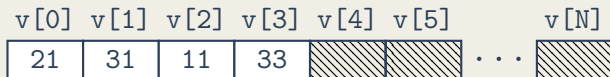
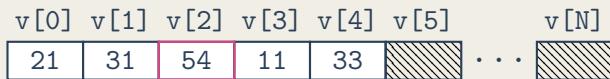
- ▶ $k=0$? n deslocamentos
- ▶ $k=n$? 0 deslocamentos

Remoção em vetor

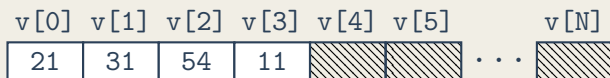
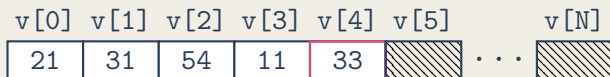
Dados um vetor $v[0..n-1]$ e um inteiro $0 \leq k < n$, remover o k -ésimo elemento

- ▶ Premissa: $n \geq 1$
- ▶ Manter ordem relativa
- ▶ Novo vetor: $v[0..n-2]$

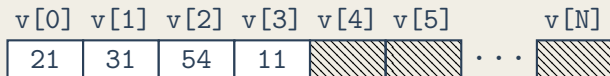
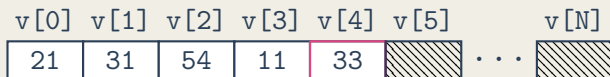
`remove(v, 5, 2);`



Remoção em vetor no final ($k=n-1$)



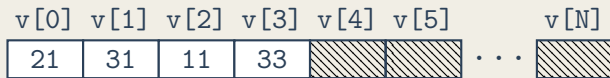
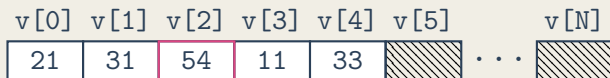
Remoção em vetor no final ($k=n-1$)



```
1 int remove_no_fim(int v[], int n, int k) {  
2     return n-1;  
3 }
```

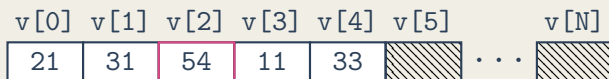
Remoção em vetor

em posição arbitrária com preservação da ordem



Remoção em vetor

em posição arbitrária com preservação da ordem



```
1 int remove(int v[], int n, int k) {  
2     int j;  
3     for (j = k; j < n - 1; j++) v[j] = v[j+1];  
4     return n - 1;  
5 }
```

Remoção em posição arbitrária

```
1 int remove(int v[], int n, int k) {  
2     int j;  
3     for (j = k; j < n - 1; j++) v[j] = v[j+1];  
4     return n - 1;  
5 }
```

Custo computacional:

► $k=0$?

Remoção em posição arbitrária

```
1 int remove(int v[], int n, int k) {  
2     int j;  
3     for (j = k; j < n - 1; j++) v[j] = v[j+1];  
4     return n - 1;  
5 }
```

Custo computacional:

- ▶ $k=0$? n deslocamentos
- ▶ $k=n-1$?

Remoção em posição arbitrária



```
1 int remove(int v[], int n, int k) {  
2     int j;  
3     for (j = k; j < n - 1; j++) v[j] = v[j+1];  
4     return n - 1;  
5 }
```

Custo computacional:

- ▶ $k=0$? n deslocamentos
- ▶ $k=n-1$? 0 deslocamentos

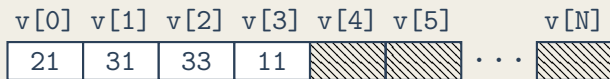
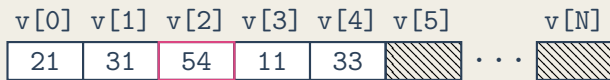
Remoção em vetor

em posição arbitrária sem preservação da ordem

$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$		$v[N]$
21	31	54	11	33		\dots	

Remoção em vetor

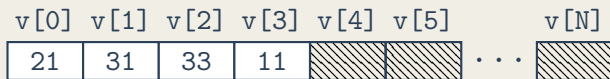
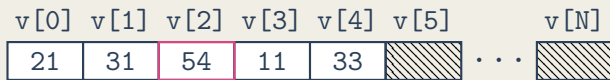
em posição arbitrária sem preservação da ordem



```
1 int remove2(int v[], int n, int k) {  
2     v[k] = v[n-1]  
3     return n - 1;  
4 }
```

Remoção em vetor

em posição arbitrária sem preservação da ordem



```
1 int remove2(int v[], int n, int k) {  
2     v[k] = v[n-1]  
3     return n - 1;  
4 }
```

1 deslocamento

Sequência de objetos com dado tamanho

```
1 #define CAP 100000
2
3 typedef struct {
4     int dados[CAP];
5     int n;
6 } lista_int;
```

Operações

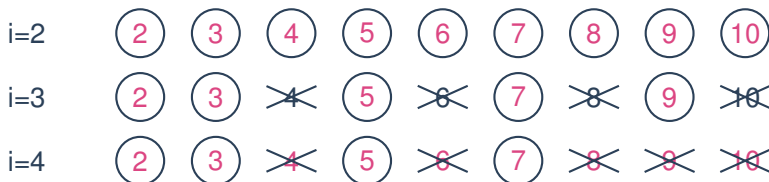
- Busca, inserção e remoção

Aplicação: Crivo de Eratóstenes

Objetivo: Determinar os números primos entre 2 e dado inteiro N

Algoritmo:

1. Liste todos os números entre 2 e N
2. Comece com $i = 0$ e repita enquanto puder:
 - 2.1 Marque todos os múltiplos de i
 - 2.2 Encontre o menor número não marcado e o chame de i (ele é primo)



Aplicação: Crivo de Eratóstenes

```
1  /* Preenche v com os n primos em [2,N]. */
2  int crivo(int v[], int N) {
3      int i,j,n = 0;
4      /* 1. Liste inteiros entre 2 e N */
5      for (i=2;i<=N;i++) v[i-2] = i;
6      /* 2. Para i=2,3,... */
7      for (i=0;i<=N;i++) {
8          /* Encontrar proximo não marcado (v[i] != 0) */
9          if (v[i]) {
10             n++; /* contador de primos */
11             /* Marque multiplos de v[i] */
12             for (j=i+v[i];j<=N;j+=v[i]) v[j] = 0;
13         }
14     }
15     /* Remova números marcados (0s) */
16     for (i=j=0;i<=N;i++)
17         if (v[i]) { v[j++] = v[i]; }
18     return n;
19 }
```


Qual a saída?

```
1  int s, *p, v[] = {1,2,3,4};  
2  
3  s = 0;  
4  
5  for (p = v; p < &v[4]; p++) s+= *p;  
6  
7  printf("%d", s);
```

Qual a saída?

```
1  int s, *p, v[] = {1,2,3,4};  
2  
3  s = 0;  
4  
5  for (p = v; p < &v[4]; p++) s+= *p;  
6  
7  printf("%d", s);
```

10

Qual a saída?

```
1 int i, *p, v[5];  
2  
3 for (i = 0; i < 5; i++) v[i] = i % 3;  
4  
5 p = v + 1;  
6  
7 while ( *p ) p++;  
8  
9 i = p - v;  
10  
11 printf ("%d %d", *p, i);
```

Qual a saída?

```
1 int i, *p, v[5];  
2  
3 for (i = 0; i < 5; i++) v[i] = i % 3;  
4  
5 p = v + 1;  
6  
7 while ( *p ) p++;  
8  
9 i = p - v;  
10  
11 printf ("%d %d", *p, i);
```

0 3

Busca em vetor ordenado

Objetivo

Recebe vetor ordenado $v[0] \leq v[1] \leq \dots \leq v[n-1]$ e valor x e determina menor posição k tal que $v[k] == x$ ou n .

```
1 int BuscaLinearComSentinela(int x, int v[], int n) {  
2     int k = 0;  
3     v[n] = x  
4     while (v[k] != x) k++;  
5     return k;  
6 }
```

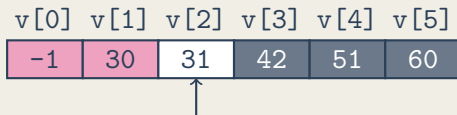
Custo computacional proporcional a tamanho n do vetor.

Algoritmo

1. Dividir vetor em dois subvetores $v[0 \dots c-1]$ e $v[c+1 \dots d]$ de tamanhos similares
2. Exclui parte que não contém x verificando “centro” $v[c]$ dos subvetores
3. Repetir até encontrar x ou terminar em subvetor vazio

BuscaBinaria(30):

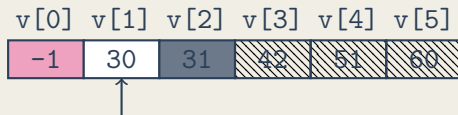
$v[0]$	$v[1]$	$v[2]$	$v[3]$	$v[4]$	$v[5]$
-1	30	31	42	51	60



Algoritmo

1. Dividir vetor em dois subvetores $v[0 \dots c-1]$ e $v[c+1 \dots d]$ de tamanhos similares
2. Exclui parte que não contém x verificando “centro” $v[c]$ dos subvetores
3. Repetir até encontrar x ou terminar em subvetor vazio

BuscaBinaria(30):



```
1 int BuscaBinaria(int x, int v[], int n) {  
2   int c, e = 0, d = n - 1;  
3   while (e <= d) {  
4     c = (e+d)/2;  
5     if (v[c] == x) return c;  
6     else if (v[c] > x) d = c-1; /* x < v[c...d] */  
7     else e = c+1; /* x > v[e...d] */  
8   }  
9   /* e > d */  
10  return n; /* busca não encontrou x */  
11 }
```

Custo computacional: Proporcional a $\log_2 n$.

Primeira abordagem: representado por vetor não-ordenado de valores não repetidos.

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
10	-1	51	37	14	54

Operações:

```
1 typedef struct {  
2   int dados[CAP];  
3   int n;  
4 } conjunto_int;
```

- ▶ Inserir elemento
- ▶ Verificar se elemento pertence ao conjunto
- ▶ Remover elemento

Aplicação: Conjuntos

Primeira abordagem: vetor não ordenado

```
1 void insere(int x, conjunto_int *C) {
2     int k = BuscaBinaria(x, C->dados, C->n);
3     if (k == C->n) C->dados[C->n++] = x;
4 }
5
6 int pertence(int x, conjunto_int *C) {
7     int k = BuscaLinear(x, C->dados, C->n);
8     return (k != C->n);
9 }
10
11 void remove(int x, conjunto_int *C) {
12     int k = BuscaLinear(x, C->dados, C->n);
13     if (k != C->n)
14         C->n = remove2(C->dados, C->n, k);
15 }
```

Aplicação: Conjuntos

Primeira abordagem: vetor não ordenado

Custo computacional de pior caso:

- ▶ Inserção: proporcional a n
- ▶ Pertencimento: proporcional a n
- ▶ Remoção: proporcional a n

Segunda abordagem: representado por vetor crescente de valores.

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
-1	10	14	37	51	54

Operações:

```
1 typedef struct {  
2   int dados[CAP];  
3   int n;  
4 } conjunto_int;
```

- ▶ Inserir elemento
- ▶ Verificar se elemento pertence ao conjunto
- ▶ Remover elemento

Aplicação: Conjuntos

Segunda abordagem: vetor crescente

```
1 void insere(int x, conjunto_int *C) {
2     int c, e = 0, d = n - 1;
3     /* encontra c tal que v[c-1] <= x < v[c] */
4     while (e <= d) {
5         c = (e+d)/2;
6         if (v[c] == x) return; /* já existe! */
7         else if (v[c] > x) d = c-1; /* x < v[c...d] */
8         else e = c+1; /* x > v[e...d] */
9     }
10    /* inserir x em k=e */
11    inserir(x, C->dados, C->n, x, e)
12 }
13
14 int pertence(int x, conjunto_int *C) {
15     int k = BuscaBinaria(x, C->dados, C->n);
16     return (k != C->n);
17 }
```

Aplicação: Conjuntos

Segunda abordagem: vetor crescente

```
1 void remove(int x, conjunto_int *C) {  
2     int k = BuscaBinaria(x, C->dados, C->n);  
3     if (k != C->n) remove(C->dados, C->n, k);  
4 }
```

Aplicação: Conjuntos

Segunda abordagem: vetor crescente

Custo computacional de pior caso:

- ▶ Inserção: proporcional a $\log n + n$
- ▶ Pertencimento: proporcional a $\log n$
- ▶ Remoção: proporcional a $\log n + n$

Aplicação: Conjuntos

Segunda abordagem: vetor crescente

Custo computacional de pior caso:

- ▶ Inserção: proporcional a $\log n + n$
- ▶ Pertencimento: proporcional a $\log n$
- ▶ Remoção: proporcional a $\log n + n$

Note: estrutura de dados afeta eficiência de operações

Matrizes (vetores bidimensionais)

```
tipo nome[tamanho1][tamanho2];
```

Sequência de vetores de mesmo tamanho e tipo

```
1  int i, j;  
2  int m[2][3];  
3  for (i = 0; i < 2; i++)  
4      for (j = 0; j < 3; j++)  
5          m[i][j] = j+3*i;  
6
```

linha\col	0	1	2
0	m[0][0]	m[0][1]	m[0][2]
1	m[1][0]	m[1][1]	m[1][2]

i\j	0	1	2
0	0	1	2
1	3	4	5

Matrizes (vetores bidimensionais)

```
tipo nome[tamanho1][tamanho2];
```

Internamente, valores são representados de maneira contígua na memória (como em um vetor unidimensional) e não como vetores de ponteiros.

$m[i][j]$ é equivalente a $*(m[i] + j)$
ou a $*(\&m[0][0] + tamanho2*i + j)$
ou a $*((int *)m[0][0] + tamanho2*i + j)$

Por conta disso, o compilador precisa conhecer a dimensão `tamanho2` quando encontra a expressão `m[i][j]` (considerando o escopo)

Matrizes (vetores bidimensionais)

```
tipo nome[tamanho1][tamanho2];
```

Sequência de vetores de mesmo tamanho e tipo

```
1 int i, j;  
2 int m[2][3];  
3 for (i = 0; i < 2; i++)  
4     for (j = 0; j < 3; j++)  
5         m[i][j] = j+3*i;  
6
```

```
1 int i, j;  
2 int m[2][3];  
3 for (i = 0; i < 2; i++)  
4     for (j = 0; j < 3; j++)  
5         *(&m[0][0] + 3*i + j) =  
6             j+3*i;
```

Funções de matrizes (vetores bidimensionais)

```
tipo nome (int v[][tamanho], int n, ...) bloco
```

argumentos: ponteiro para matriz e número de linhas

tamanho é uma **constante**

```
1  int maximo (int m[][3], int n) {  
2      int i, j, maior = m[0][0];  
3      for (i=0; i < n; i++)  
4          for (j=0; j < 3; j++)  
5              if (m[i][j] > maior) maior = m[i][j];  
6      return maior;  
7  }  
8
```

Funções de matrizes (vetores bidimensionais)

```
tipo nome (int v[][tamanho], int n, ...) bloco
```

- ▶ Omitir tamanho gera erro
- ▶ Usar ponteiro tipo *nome gera erro

```
1 int maximo (int m[][ ], int n); /* ERRO */
2 int maximo (int *m, int n); /* ERRO */
3 int maximo (int ( *m )[3], int n); /* OK */
4
```

*Parênteses são importantes no último caso pois [] tem precedência sobre **

Vetores de ponteiros

```
tipo *nome[tamanho]
```

São declarados e definidos de acordo.

Uso: quando queremos ter vetores de vetores com diferentes dimensões

```
1 int main(int argc, char *argv[]);  
2 /* argv é vetor de ponteiros */
```

- ▶ Exercícios 3A-3E
- ▶ 6/9, 8/9: Recesso
- ▶ 11/9: Prazo para entrega dos exs. 1, 2 e 3.