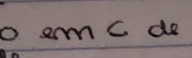
- **Estrutura de Dados**
  - Implementação em C de uma Fila de Espera sobre um array, com circularidade
    - enqueue (acrescenta um elemento na última posição da fila)
    - dequeue (remove - e devolve - o primeiro elemento da fila)

```
typedef struct queue {
    int inicio, tam;
    int MAX
    int *valores;
} QUEUE;
```

- Array com circularidade

p.e. enqueue



i = 2
tam = 3
MAX = 5

posição a ser colocado o próximo elemento: (2+3) % 5 = 0

- Array Dinâmico
  - quando o array enche (tamanho = capacidade) realocamos para um array de capacidade dupla

```
malloc (2*q → capacidade * sizeof(int))
calloc (2*q → capacidade * sizeof(int))
```

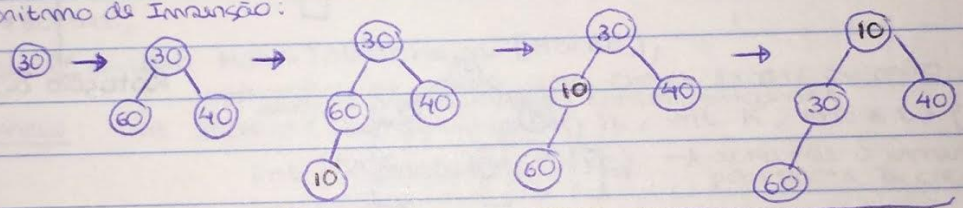
(MAX passa para "capacidade")

- Filas com Prioridade e "Heaps"

O tipo abstrato de dados
- DICIONÁRIOS
- BUFFERS:
  - FIFO (Queue)
  - LIFO (Stacks)
  - Filas com prioridades

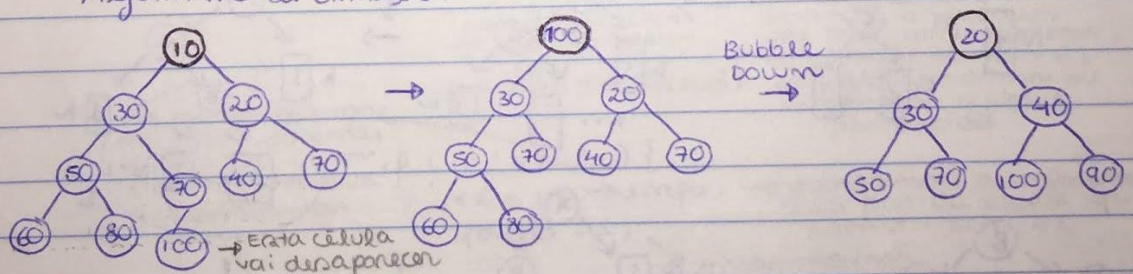
- (HEAPS) (min-heap)
  - Invariante de ordem: o valor associado a cada nó é inferior ou igual ao valor de todos os seus descendentes
  - Invariante de forma: só o último nível é que pode estar incompleto e é preenchido da esquerda para a direita

Algoritmo de Inserção:

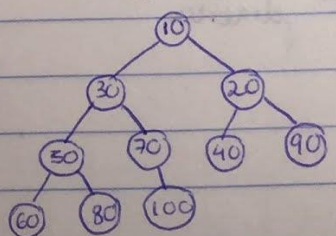


Bubble Up

Algoritmo de Extração:



Implementação Física



0	1	2	3	4	5	6	7	8	9
10	30	20	50	70	40	90	60	80	100

→ V[i] = [10, 30, 20, 50, 70, 40, 90, 60, 80, 100]

- Árvores AVL
- Tipos Abstratos de Dados e Estruturas de Dados

Árvore
Árvore Binária
Árvore Binária de Procura (BST) (inorder)

totalmente desiquilibrada | equilibrada

Procura: $T(N) = \Omega(1), \Theta(N)$ | Procura: $T(N) = \Omega(1), O(\log N)$

Inserção: $T(N) = \Omega(1), O(N)$ | Inserção: $T(N) = \Theta \log N$

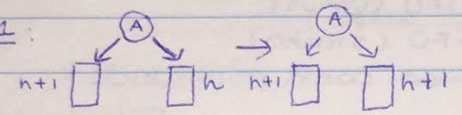
AVL

↳ As alturas da sub-árvore da esquerda e da sub-árvore da direita diferem no máximo numa unidade: $|he - hd| \leq 1$

Inserção, Remoção e Procura: $T(N) = \Theta(\log N)$

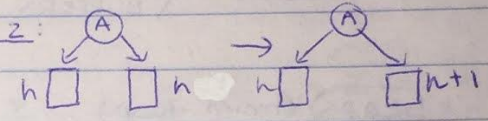
Algoritmo de Inserção:

caso 1:

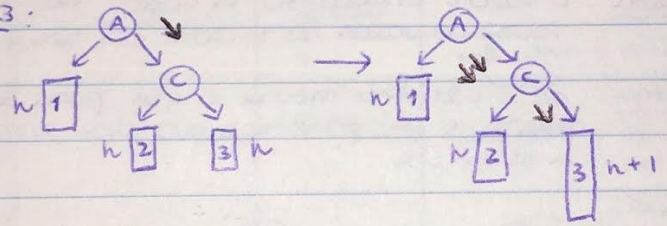


(simétrico para o esquerda)

caso 2:

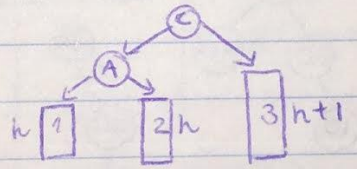


caso 3:



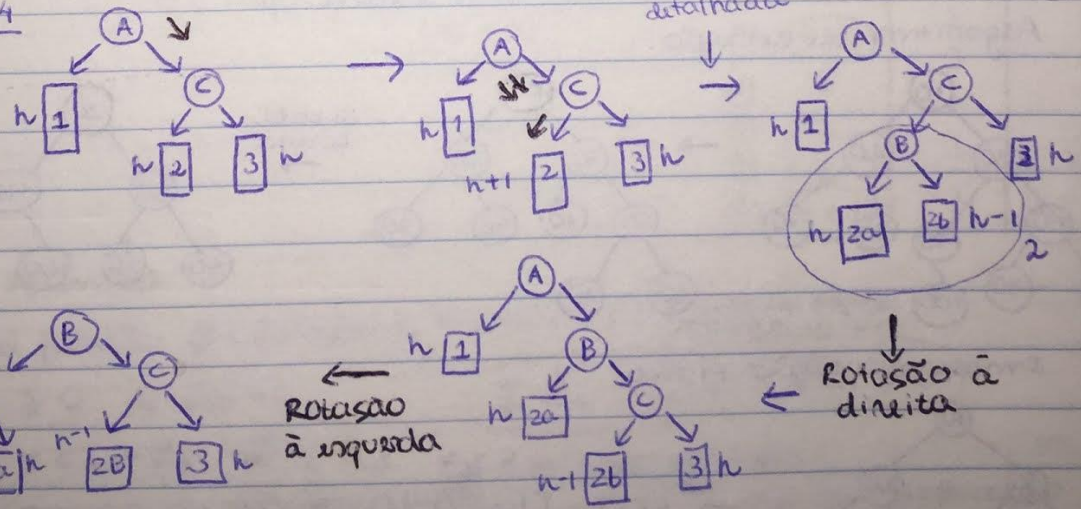
(↓↓)

→ Temos de reajustar esta árvore



← Rotação à esquerda

caso 4



análise mais detalhada

Rotação à direita

Rotação à esquerda

• Tabelas de Hash

par (K, i)
↓ ↓
chave informação

função hash - converte cada chave num índice de um array
array onde se armazena a informação

(por exemplo, para encontrar um elemento no array, eles têm que nos dar uma chave, usamos a função hash para descobrir o índice correspondente e depois vamos buscar a informação aí contida)

- Há maneiras diferentes de resolver problemas como:
  - analisar se uma célula do array está a armazenar algo significativo ou não
  - resolver colisões (quando dois pares, depois de aplicar a função hash a ambas as suas chaves, têm o mesmo índice).
- Assumimos: HSIZE (tamanho do array)
  int hash (int Key, int size)

CLOSED ADRESSING

Forma de resolver colisões → os pares que colidem numa posição são armazenados (p.e.) numa lista ligada

```
typedef struct bucket {
    int Key; int info;
    struct bucket *next;
} *Bucket;
typedef Bucket HashTableChain [HSIZE];
```

Exemplo: (verifica se vamos cuja chave existe ou não; se sim, devolve a informação correspondente)

```
int lookup (HashTableChain h, int K, int *i) {
    int p = hash (K, HSIZE);   → acha-nos o endereço correspondente à chave K
    int found;
    Bucket it;
    for (it = h[p] ; it != NULL && it -> Key != K; it = it -> NEXT)
        (começa a procurar no endereço "calculado" (na lista desse "balde"); continua a percorrer essa lista, correspondente ao endereço da chave K, até esta terminar ou encontrar a chave pretendida; avança na lista)
    (se não chegarmos ao final da lista ou não encontrarmos a chave que queríamos)
    if (it != NULL) {
        *i = it -> info;   → guardamos a informação do par da chave que procurámos em *i
        found = 1;
    }                      → encontrámos
    else found = 0;        → não encontrámos
    return found;
}
```

Bucket it
it = h[p]
it != NULL && it->key != K
it = it->next

Bucket *it
it = h+p
*it != NULL && (*it)->key != K
it = &((*it)->next))

OPEN ADRESSING

FORMA de resolver colisões → probing: é calculada uma posição alternativa para armazenar o par que causou a colisão

A QUE SAI NOS TESTES ←

```
#define STATUS_FREE 0
#define STATUS_USED 1

typedef struct bucket {
    int status,
    int key; int info;
} Bucket;

typedef Bucket HashTable[HSIZE];
```

LINEAR PROBING

↳ se a posição determinada através da chave estiver ocupada, avança para a próxima

Exemplo

```
int update (HashTable h; int K, int i) {
    int p = find_probe (h,K); → encontra posição livre
    int r;
    if (p<0) → não há nenhum lugar livre na tabela (tabela cheia)
        r=0;
    else if (h[p].key == K){ → encontrou uma célula com uma key igual
        h[p].info = i; → atualiza a info
        r=1;
    else{ → é uma key nova (célula vazia)
        h[p].status = STATUS_USED;
        h[p].key = K;
        h[p].info = i;
        r=2
    }
    return r;
}
```

QUADRATIC PROBING

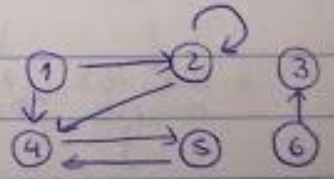
↳ para evitarmos muitas colisões como no "linear probing" (devido à criação de clusters), vai-se aumentando o espaçamento entre probes sucessivos.

• Algoritmos sobre Grafos

• Representação de Grafos em computador

GRAFOS ORIENTADOS

(V, E)
↓ vértices ↘ arestas



grau de entrada de V: número de arestas com destino a V

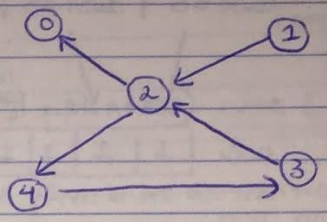
grau de saída de V: número de arestas com origem em V

número máximo de arestas: $E \leq V^2$

Representação em computador de Grafos orientados
- Matriz de Adjacências
- Lista de Adjacências ← É SEMPRE ESTE TIPO QUE SAI NOS TESTES

① Matriz de adjacência

origem \ destino	0	1	2	3	4
0	0	0	0	0	0
1	0	0	1	0	0
2	1	0	0	0	1
3	0	0	1	0	0
4	0	0	0	1	0



pesos não podem ser nulos (0 está reservado para representar quando não há arestas)

→ para grafos pesados, em vez do 1 punha-se o peso das arestas

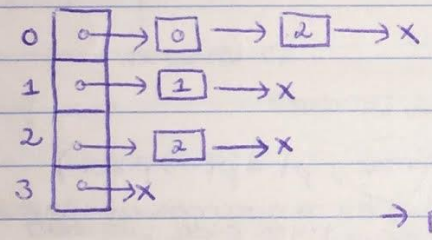
→ se quiséssemos fazer para um não orientado, bastava espelhar a tabela (mas podíamos apenas ficar com metade da matriz triangular) ↓ (evita-se assim redundância)

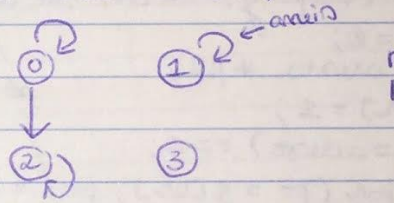
```
#define MAX 100
typedef char (int) graphM [MAX][MAX]
```
(int) pesados

$M(V,E) = \Theta(V^2)$ (memória)

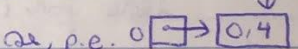
$T_{adj}(V,E) = \Theta(1)$

② Lista de adjacência (não dispendiosamos espaço a guardar todos os 0's)





Pesos já podem ser nulos

Peso


→ para grafos pesados, ponha-se, p.e. 0 → 0,4

→ para grafos não orientados os anéis desapareciam todos, p.e., neste caso o 2 também apontava para 0 (mais redundância)

```
#define MAX 100
struct edge {
    int dest;
    (int weight;) pesados
    struct edge *next;
};
typedef struct edge * GraphL [MAX];
```

$M(V,E) = \Theta(V+E)$

$T_{adj}(V,E) = \mathcal{O}(V)$ → obriga a uma travessia

- Algoritmos de Travessia de Grafos

Árvores / Florestas

- grafos acíclicos orientados em que todos os vértices têm grau de entrada 0 ou 1.
- $(U,V) \in E \rightarrow U$ é pai de V
- vértice com grau de entrada 0 → raiz
- uma única raiz - árvore

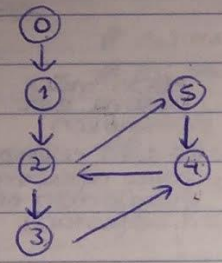
Travessia de Grafos

↳ visitar todos os vértices alcançáveis a partir de um inicial, não passando mais do que uma vez em cada um deles. Com isto é criada uma árvore de travessia.

↗ Todos os vértices adjacentes a um vértice v são visitados imediatamente a seguir a v

TRAVESSIA EM PROFUNDIDADE (Depth-First)

Não pode ser utilizada para < calcular distâncias entre vértices
calcular caminhos mais curtos entre vértices



or=0 dest=5	or=1 dest=5	or=2 dest=5	or=3 dest=5	or=4 dest=5	or=5 dest=5

mas lugares que ficam por 1 no con[]

con	1	1	1	1	1	1
	0	1	2	3	4	5

ant	-1	0	1	2	3	2
	0	1	2	3	4	5

→ PAIS
→ SUCESSORES

Não há sucessores do 4 com 0 no array con[]
↓
Não há sucessores do 3 com 0 no array con[]
↓
chega ao 2 e vai para o seu sucessor 5

```
int procura (Grafo g, int or, int dest, int ant[]) {
    int con[N], i;                          todos os vértices estão por visitar
    for (i=0; i<N; i++) { con[i] = 0; ant[i] = -1 }
    return (procuraAux(g, or, dest, con, ant));      ↳ inicializa a árvore com as raízes a -1
}
```

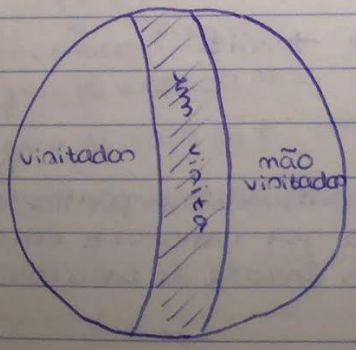
```
int procuraAux (Grafo g, int or, int dest, int con[], int ant[]) {
    int r = 0;
    struct aresta *pt;
    con[or] = 1;
    if (or == dest) r = 1;
    else for (pt = g[or]; pt != NULL && r==0; pt = pt->prox)
                                                  or != dest
        if (con[pt->dest] == 0) {  → se o sucessor ainda não tiver sido visitado
            ant[pt->dest] = or;
            procuraAux(g, pt->dest, dest, con, ant);
        }                          "avança-se na origem"
    return r;
}
```

TRAVESSIA EM LARGURA (Breadth-First)

↳ Todos os vértices à distância K de v são visitados antes de qualquer vértice à distância K+1

→ a árvore de travessia construída contém todos os caminhos mais curtos com origem em v e destino em cada um dos vértices alcançáveis a partir de v

↳ o caminho mais curto entre v e d dá a distância entre eles



0 - não visitado - BRANCO
1 - em visita - CINZENTO
2 - visitado - PRETO

```
#define BRANCO 0;
#define CINZENTO 1;
#define PRETO 2;

int travessiaBF (Grafo g, int ori, int ant[]) {
    int onda[N];
    struct aresta *pt;
    r = 0;
    int ini, fim; ini = fim = 0;
    int cor[N];
    int i;
    for (i=0; i<N; cor[i++] = BRANCO) ant[i] = -1;
    onda[fim++] = ori;
    cor[ori] = CINZENTO;
    while (ini != fim) {
        v = onda[ini++];
        r++;
        cor[v] = PRETO;
        for (pt=g[v]; pt != NULL; pt = pt->prox)
            if (cor[pt->dest] == BRANCO) {
                onda[fim++] = pt->dest;
                cor[pt->dest] = CINZENTO;
                ant[pt->dest] = v;
            }
    }
    return r;
}
```

inicializa a árvore com as raízes a -1

fim - nº de vértices que puseram na onda

ini - nº de vértices que tiraram da onda

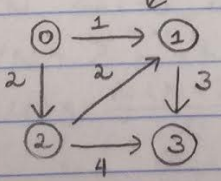
r - nº de vértices atravessados

Floyd-Warshall: calcula o caminho mais curto entre todos os vértices e guarda os resultados numa matriz

↳ serve para percorrer o grafo

origem \ destino	0	1	2	3
0	-1	1	2	4
1	-1	-1	-1	3
2	-1	-1	2	4
3	-1	-1	-1	-1

Dijkstra SP: calcula o caminho mais curto do vértice a 0 para todos os vértices do grafo g



pais

0	1	2	3
-1	0	0	1

pesos

0	1	2	3
-1	1	2	4

(pesos dos caminhos mais curtos)

Achar o percurso do caminho mais curto de 0 = 0 até ao vértice mais distante → Máximo dos pesos: 4 → vértice 3 → Ir ao array pais e ir ver o caminho de 0 a 3 (antecessor de 3 é 1 → antecessor de 1 é 0)

Caso médio

possíveis execuções r < custo c_r / probabilidade p_r

$$\overline{T}(N) = \sum_r p_r * c_r$$

Exemplo:
(número de acessos ao array)!!

```
int lsearch (int e, int N, int v[N]) {
    int i;
    i = 0;
    while ((i<N) && (v[i]<e))
        i++;
    if ((i=N) || (v[i]!=e)) return (-1);
    else return i;
}
```

(v ordenado)

MC: v[i]<e é FALSO, LOGO TEMOS APENAS 2 ACESSOS AO ARRAY < v[i]<e ✗ / v[i]!=e

$$T(N) = 1 + 1 = 2$$

PC: O CICLO SÓ ACABA QUANDO i>=N
i.e. O ELEMENTO É MAIOR QUE TODOS DO ARRAY

$$T(N) = \underbrace{\sum_{i=0}^{N-1}}_{\text{nº de ciclos realizados (nº de "v[i]<e")}} 1 + 1 = N + 1$$

N = 2, e = 3

1	2
0	1

while (0<2 && 1<3) ✓
while (1<2 && 2<3) ✓
while (2<2 ———) ✗ (x)
... v[i]!=e

N + 1 / 2 + 1 acessos

Custo médio: como os arrays não uniformemente distribuídos e o valor a procurar é aleatório, podemos assumir que o ciclo pode fazer, com igual probabilidade, de 0 a N-1 iterações

Exemplo:

1	2	3	4	5
0	1	2	3	4

N = 5, e = 3

while (0<5 && 1<3) ✓
while (1<5 && 2<3) ✓
while (2<5 && 3<3) ✗
... v[i]!=e

(K) 2 ciclos → 2+1+1 = 4 acessos ao array (K+2)

o ciclo pode parar de 0 a N-1 → N possibilidades (1/N de probabilidade)

$$\overline{T}(N) = \underbrace{\sum_{i=0}^{N-1}}_{\text{nº de ciclos}} \underbrace{(1/N)}_{\text{probabilidades}} * \underbrace{(i+2)}_{\text{custo total de acessos}}$$

0	1	2	3
0+2	1+2	2+2	3+2
2	3	4	5

$$= 1/N \times \sum_{i=0}^{N-1} (i+2)$$

$$= 1/N \times \sum_{i=2}^{N+1} i$$

$$\sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$$

$$= 1/N \times \frac{N(N+3)}{2}$$

Exemplo:

```
int strcmp (char a1[], char a2[])
    int i;
    for (i=0; a1[i] && a2[i] && a1[i] == a2[i]; i++)
    return (a1[i] - a2[i]);
}
```

(considerando o nº de comparações)

MC: (não diferentes logo no primeiro elemento) $T(N) = 1$

PC: (strings iguais) $T(N) = N$

Caso médio:

Alfabeto : 26

custo	Probabilidade
1	$1 - \frac{1}{26} = \frac{25}{26}$ → ser diferente no 1º elemento
2	$\frac{25}{26} \times \frac{1}{26}$
3	$\frac{25}{26} \times \left(\frac{1}{26}\right)^2$
⋮	⋮
N	$\frac{25}{26} \times \frac{1}{26^{N-1}} = \frac{25}{26^N}$

$$\sum_{i=1}^{\infty} i c^i = \frac{c}{1-c}, \quad |c| < 1$$

$$\overline{T}(N) = \sum_{i=1}^{N} i \times \frac{25}{26^i} = 25 \sum_{i=1}^{N} i \times \left(\frac{1}{26}\right)^i = 25 \times \frac{1/26}{1 - 1/26} = \frac{\frac{25}{26}}{\frac{25}{26}} = 1$$

Exemplo: (vetor com os bits de um número)

```
int inc (int b[], int N){
    int i, r = 0;
    i = N-1;
    while ((i>=0) && (b[i]==1)) {
        b[i] = 0;
        i--;
    }
    if (i>=0) b[i] = 1;
    else r = 0;
    return r;
}
```

ex: 1111 ✗

0(1)(1)(1) → 1000 (PC)

N = 4

N − 1 elementos

1000(0) → 1001 (MC)

(nº de bits alterados)

M.C: $T(N) = 1$ (muda 1 bit)

PC: $T(N) = N$ (muda todos os bits)

N = 4

1111, 0111

Caso médio:

custo	Probabilidade
1	$1/2^{(0)}$
2	$1/2^{(1)} \times 1/2^{(0)}$
3	$1/2^{(1)} \times 1/2^{(1)} \times 1/2^{(0)}$
⋮	⋮
K	$(1/2)^K$

$$\overline{T}(N) = \left(\sum_{i=1}^{N} i \times \left(\frac{1}{2}\right)^i \right) + N \times \frac{1}{2^N} = \frac{1/2}{1 - 1/2} + \frac{N}{2^N}$$

$$= 1 + \frac{N}{2^N}$$

custo 3: 111

p/ N=3: custo 1: 0

custo 2: 01

custo 3: 011

Exemplo:

```
int prod (int x, int y) {
    int r;
    r = 0;
    while (x > 0) {          ---> nº de iterações depende de x
        r = r + y; x = x - 1;
    }
    return r;
}
```

(considerando o nº de adições feitas à variável r)

x: N bits

MC: menor valor de x
PC: maior valor de y

$$T(N) = 2^{N-1}$$

Exemplo:

```
int bsearch (int x, int N, int v[N]) {
    int i, o, m;
    i = 0; o = N - 1;
    while (i < o) {
        m = (i + o) / 2;
        if (v[m] == x) i = o = m;        --> MC
        else if (v[m] > x) o = m - 1;
        else i = m + 1;
    }
    if ((i > o) || (v[i] != x)) return -1;
    else return i;
}
```

v: | 0 | 1 | 2 | 3 | N = 4, x = 6

i = 0, o = 4 − 1 = 3;

(0 < 3) ✓ m = (0 + 3) / 2 = 1 if (v[1] == 6) ✗ else if (v[1] > 6) ✗ else i = 1 + 1 = 2	3 − 0 = 3
(2 < 3) ✓ m = (1 + 3) / 2 = 2 if (v[2] == 6) ✗ else if (v[2] > 6) ✗ else i = 2 + 1 = 3	3 − 2 = 1

x: o número que queremos encontrar é menor) avançamos com o o para "esquerda"

caso contrário avançamos com o i para a direita

(consideremos o número de acessos ao vetor) → determinado pelo número de iterações do ciclo

MC: o x está no 1º elemento do array acedido

T = 1 (apenas 1 acesso ao array)

PC: o x não está no array

i < o

||

o − i > 0 → a cada ciclo o − i passa a metade

Logo o número máximo de iterações é $\log_2 N$.

(nº de vezes que consegui - mos dividir o array a meio)

$$T = \log_2 N$$

Caso médio: o elemento existe com igual probabilidade em qualquer posição do array (1/N)

x = 4

v: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | N = 7
0 1 2

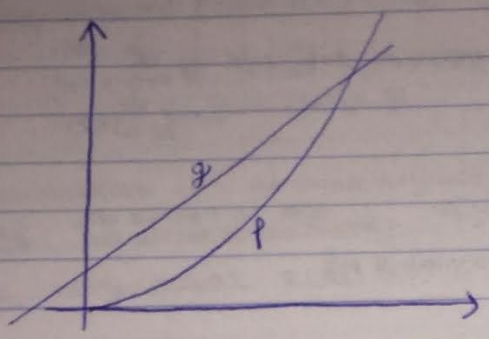
??

custo (nº de ciclos)		probabilidade
(1 possibilidade do ciclo executar 1 vez)	1	1/N
(2 possibilidades do ciclo executar 2 vezes)	2	2/N
(4 possibilidades do ciclo executar 3 vezes)	3	4/N
⋮	⋮	⋮
(2^{K-1} possibilidades do ciclo executar K vezes)	K	$2^{K-1}/N$

$$\overline{T}(N) = \sum_{K=1}^{\log_2 N - 1} K \times \frac{2^{K-1}}{N}$$

Análise Assintótica

Comparam taxas de crescimento do "custo" da função



$g \in O(f)$ e $f \notin O(g)$ ← f delimita o crescimento de g

$f \in \Theta(g)$ ↓ se tiverem o mesmo grau de crescimento

PC: $g \in O(f)$ ↓ limite superior

e

MC: $f \in \Omega(g)$ ↓ limite inferior

$f(N) = k_1 N^2 + k_2 N + k_3 = \Theta(N^2)$

↳ TEMPO QUADRÁTICO

Definições Recursivas

Exemplo:

```
void bubble_sort (int N, int v[N]) {
    int i, j;
    for (i=N-1; i>1; i--)
        for (j=0; j<i; j++)
            if (v[j] > v[j+1])
                swap (v, j, j+1);
}
```

c = comparações
w = nº de escritas no array

depende do tamanho do array

→ também depende do conteúdo do array

Vamos considerar o nº de swaps feitos para determinar o MC e PC:

MC: v[j] nunca é maior que v[j+1] (array ordenado)

PC: array desordenado completamente (verifica-se sempre a condição do if)

MC: $T(N) = \underbrace{\sum_{i=2}^{N-1}}_{\text{aqui como o ciclo decrementa temos que "inverter"}} \left(\sum_{j=0}^{i-1} c \right) = \sum_{i=2}^{N-1} (i \times c) = c \times \sum_{i=2}^{N-1} i$

de $i = N-1$ até $i > 1$ $i--$

$=$ de $i = 2$ até $i = N-1$ $i++$

$\sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$

$= c \times \frac{(2+N-1) \times (N-2+1)}{2}$

$= c \times \frac{(N+1)(N-2)}{2}$

$= c \times \frac{N^2 - 2N + N - 2}{2}$

$= \Theta(N^2)$

PC: $T(N) = \sum_{i=2}^{N-1} \left(\sum_{j=0}^{i-1} c + 2w \right) = \cdots = \Theta(N^2)$

Logo o tempo de execução em geral da função $= \Theta(N^2)$

$T(^n/_d)$ altura $= \log_d(n) + 1$

Exemplo : void merge_sort (int N, int v[N])

```
int m;
if(N>1){
    m = N/2;
    ① merge_sort (m, v)
    ② merge_sort (N-m, v+m);
    ③ merge (N, v, m);
}
```

↳ $T_{merge}(N) = 2 \times N = \Theta(N)$

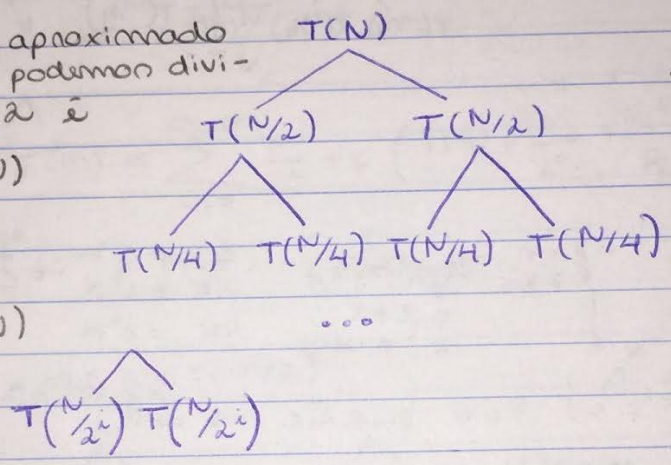
$$T_{merge\_sort} = \begin{cases} 1 & , N \leq 1 \\ 2N + T(N/2) + T(N/2) & , N > 1 \\ \text{③} \quad \text{②} \quad \text{①} \end{cases}$$

O número aproximado de vezes que podemos dividir N por 2 é

$\log_2(N)$

↓

$i = \log_2(N)$



$T(0) = 1$

2^0 $T(1) = 1$

2^1 $T(2) = 2 \times 2 + T(\frac{2}{2}) \times 2$

$= 2 \times 2 + 1 \times 2$

2^2 $T(4) = 2 \times 4 + T(\frac{4}{2}) \times 2$

$= 2 \times 4 + 2 \times (2 \times 2 + 2)$

$= 2 \times 4 + 2 \times 4 + 4$

...

$T(2^i) = \underbrace{2 \times 2^i + \cdots + 2 \times 2^i}_{i \text{ vezes}} + 2$

$= (2^i \times 2) \times i + 2^i$

$= 2^i \times (2i) + 2^i$

$= 2^i (2 \times i + 1)$

$T(N) = T(2^{\log_2(N)})$

$= 2^{\log_2(N)} \times ((2 \times \log_2(N)) + 1)$

$= N \times (2\log(N) + 1)$

$= 2N\log(N + 2N = \Theta(N \times \log_2(N))$

(ou $T(N) = \sum_{i=0}^{\log_2 N} (2N) = \underbrace{(\log_2 N + 1)}_{\text{nº de recorrências}} \times 2N = 2N + 2N\log_2 N = \Theta(N \cdot \log_2 N)$)

$2N \rightarrow 2N$

$N + N \rightarrow 2N$

$\frac{N}{2} + \frac{N}{2} + \frac{N}{2} + \frac{N}{2} \rightarrow 2N$

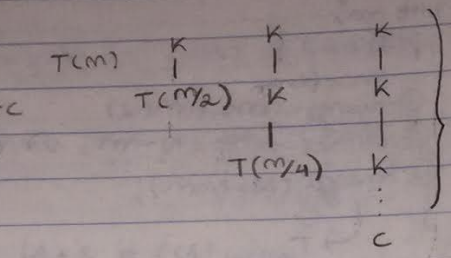
Exemplos:

- $T(m) = K + T(m/2)$

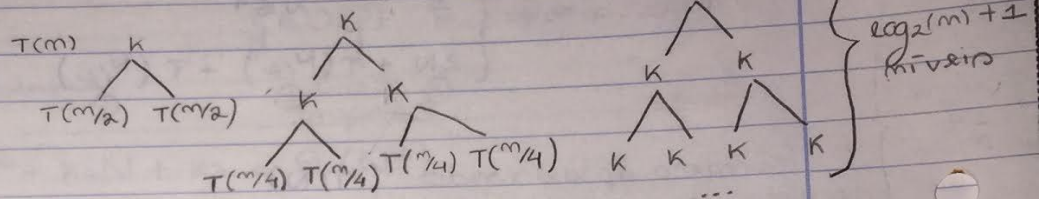
$= K \times (\log_2(m)+1) + c$

$= K \log_2(m) + K + c$

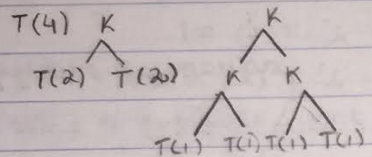
$= \Theta(\log_2(m))$

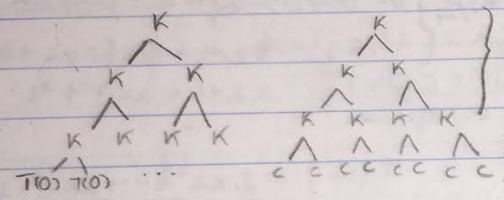


- $T(m) = k + 2 \times T(m/2)$



Ex:





$\log_2(4)+1$

$= 2+1$

$= 3$ níveis

$1K = 2^0 K$

$2K = 2^1 K$

$4K = 2^2 K$

$\rightarrow \sum_{i=0}^{\log_2(N)} 2^i K$

$8c = 2^3 c \rightarrow 2^{\log_2(m)+1}$

$\sum_{i=0}^{m} a^i = \frac{a^{m+1}+1}{a-1}$

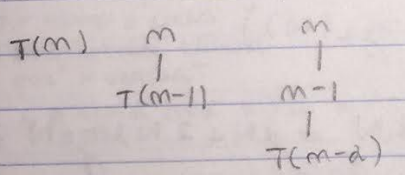
$T(m) = \sum_{i=0}^{\log_2(m)} 2^i K + 2^{\log_2(m)+1}$

$= K \times \frac{2^{\log_2(m)+1}+1}{2-1} + 2^{\log_2(m)+1}$

$= K \times 2 \times 2^{\log_2(m)} + K \times 1 + 2 \times 2^{\log_2(m)}$

$= 2Km + K + 2m = \Theta(m)$

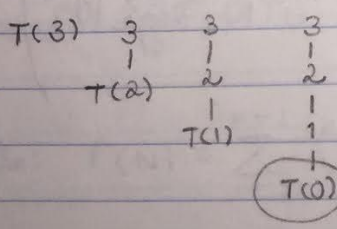
- $T(m) = m + T(m-1)$



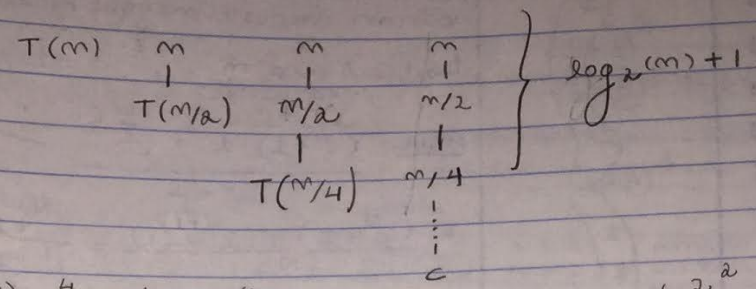
$\sum_{i=1}^{N} i \rightarrow \sum_{i=0}^{m} i = \frac{m(m+1)}{2}$

$T(m) = \sum_{i=0}^{m} i + c$

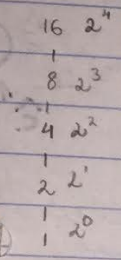
$= \frac{m(m+1)}{2} + c = \frac{m^2+m}{2} + c = \Theta(m^2)$

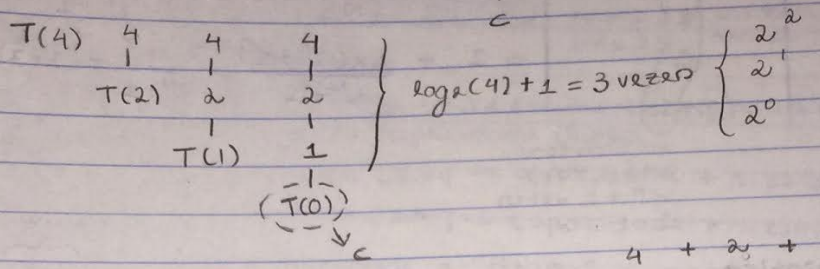


• $T(n) = n + T(n/2)$



$\log_2(16) = 4$



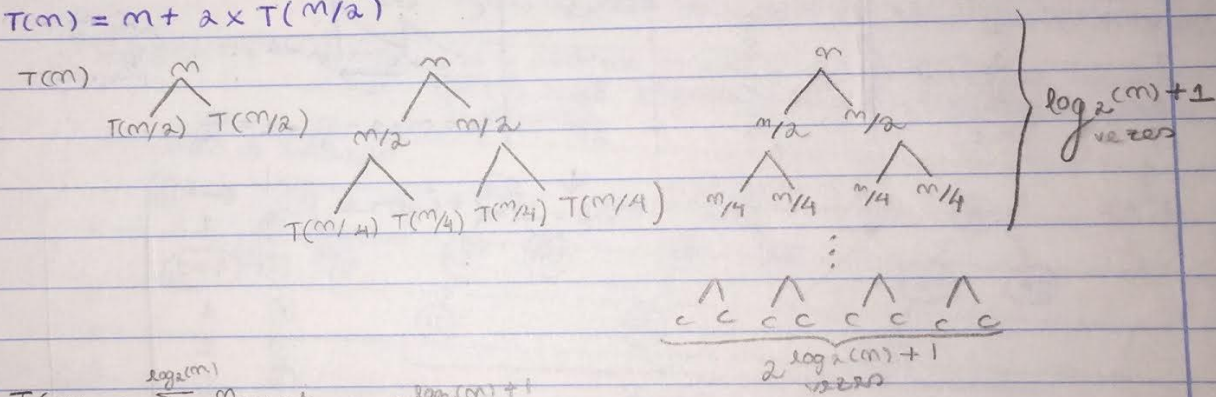


$$T(n) = \sum_{i=0}^{\log_2(n)} \frac{n}{2^i} + c \quad \left(T(4) = \frac{4}{2^0} + \frac{4}{2} + \frac{4}{2^2} + c \right)$$

$$\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$$

$$\rightarrow = n \times \left(2 - \frac{1}{2^{\log_2(n)}} \right) = n \times \left(2 - \frac{1}{n} \right) = 2n - 1 = \Theta(n)$$

• $T(n) = n + 2 \times T(n/2)$



$$T(n) = \sum_{i=0}^{\log_2(n)} \frac{n}{2^i} \times 2^i + 2^{\log_2(n)+1} \times c$$

$$= n \times (\log_2 n + 1) + 2 \times n \times c$$

$$= n \log_2 n + n + 2nc = \Theta(n \log_2 n)$$

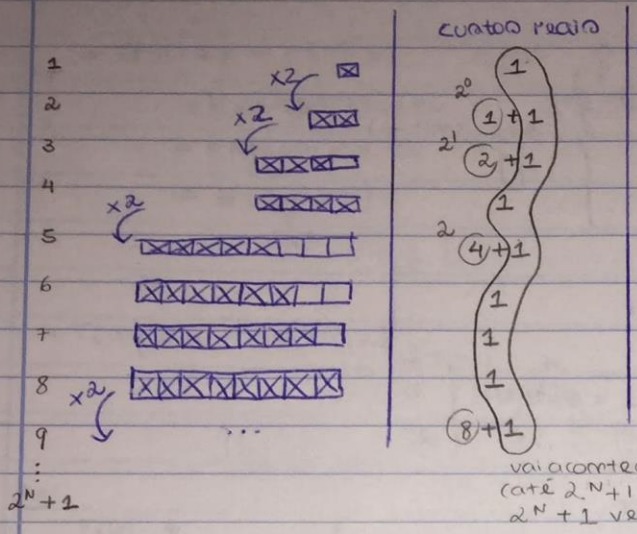
Análise Amortizada

cop – custos reais das operações $\quad \sum cop \leq \sum \hat{cop}$

$\hat{cop}$ – custos amortizados das operações

① **Análise Agregada**

custos reais de push
- MC: 1 (array não cheio)
- PC: N + 1 (array cheio – realocar tudo e acrescentar 1)



Fazendo a média normal: $\frac{24}{9} \simeq 2.5$

Mas 2.5 diferencia muito comparativamente aos casos em que temos que realocar o array

↳ PC: A cabar (p.e. 4+1 ou 8+1)
custo elevado

Média: $\frac{\overbrace{(2^N+1)\cdot 1}^{1's} + \sum_{i=0}^{N} 2^i}{2^N+1} \rightarrow \hat{c} = \frac{\sum c_i}{N}$

$= \frac{(2^N+1)+(2^{N+1}-1)}{2^N+1} = \frac{2^N+1}{2^N+1} + \frac{2^N \times 2 - 1}{2^N+1}$ (2 − 3 = −1)

$= 1 + \frac{2\times(2^N+1)}{2^N+1} - \frac{3}{2^N+1} = 1+2-\frac{3}{2^N+1} \simeq 3$

$\hat{c} = 3$

② Método contabilístico

crédito — débito

$\sum c_i \leq \sum \hat{c}_i \iff \underbrace{\sum \hat{c}_i - \sum c_i}_{\text{saldo}} \geqslant 0$

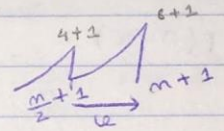
	Débito (c)	crédito ($\hat{c}$)	saldo
1	1	5	4
2	1+1	5	7 (10−3)
3	2+1	5	9 (15−6)
4	1	5	13
5	4+1	5	13
6	1	5	17
7	1	5	21
8	1	5	25
9	8+1	5	21
⋮			
2^N+1			

5 foi uma estimativa à mão

$S_i = S_{i-1} + \hat{c}_i - c_i$ ($S_0 = 0$)

→ com $\hat{c}=5$, a equação verifica-se mas o saldo é demasiado grande

sabemos que quando aparece uma operação "cara" temos de ter no saldo $N+1$



(K) $\frac{m}{2} = m \iff k = 2$

↓ poupança

3

↓	(...)	(...)	(...)
④	1	3	⑤
⑤	4+1	3	3
⑥	1	3	5
⑦	1	3	7
⑧	1	3	⑨
⑨	8+1	3	3

③ Método do Potencial

Definimos uma função ϕ (função de Potencial) :: Estado → Float

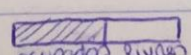
- $\forall \phi_s \geqslant 0$ e $\phi_0 = 0$

↳ $\hat{c}_i = c_i + \phi_i - \phi_{i-1}$

$\sum \hat{c}_i - \sum c_i = \phi_n \geqslant 0$

ϕ = ocupados − livres

1) caso sem duplicação



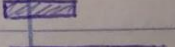
$\hat{c}_{push} = 1 + \phi_{depois} - \phi_{antes}$

$= 1 + (\text{ocup}_{dep.} - \text{Livres}_{dep}) - (\text{ocup}_{ant} - \text{Livres}_{ant})$

$= 1 + (\text{ocup}_{ant} + 1 - (\text{Livres}_{ant} - 1) - (\text{ocup}_{ant} - \text{Livres}_{ant})$

$= 1 + 1 + 1 = 3$

2) caso com duplicação



$\hat{c}_{push} = (N+1) + \phi_{depois} - \phi_{antes}$

$= (N+1) + (\text{ocup}_{dep} - \text{Livres}_{dep}) - (\text{ocup}_{ant} - \text{Livres}_{ant})$

$= (N+1) + (N+1) - (N+1) - N$

$= 3$