MC202 - Estruturas de Dados

Guilherme P. Telles

IC

3 de Outubro de 2019

Avisos

- Estes slides contêm erros.
- Estes slides são incompletos.
- Estes slides foram escritos usando português anterior à reforma ortográfica de 2009.

Situações triviais em estruturas de dados

- Se o volume de dados é muito pequeno então colocar os dados em um array e fazer uma busca seqüencial provavelmente é uma boa solução.
  - Por exemplo, registrar 20 pessoas por CPF.
  - As operações não levarão tempo constante, mas são tão poucos dados que qualquer solução mais sofisticada não será mais eficiente ou terá um ganho de eficiência irrisório.
- Se a freqüência de acesso aos dados é muito pequena (processamentos anuais, bienais etc.) então colocar os dados em um array talvez seja uma boa solução.

Parte I

Seqüência, fila, pilha, deque

Seqüência

- Uma *seqüência* (ou lista) é uma sucessão de elementos que pode ter repetições.
- A posição relativa entre os elementos da seqüência é importante (isso não implica que eles estejam em ordem segundo algum atributo).
- As operações típicas sobre uma seqüência S são:
  - retornar o i-ésimo elemento de S – $\textsc{Get}(S, i)$.
  - retornar o primeiro elemento de S – $\textsc{Get-Head}(S)$.
  - retornar o último elemento de S – $\textsc{Get-Tail}(S)$.
  - adicionar o elemento x ao início de S – $\textsc{Inject}(S, data)$.
  - remover e retornar o primeiro elemento de S – $\textsc{Eject}(S)$.
  - adicionar o elemento x ao fim de S – $\textsc{Push}(S, data)$.
  - remover e retornar o último elemento de S – $\textsc{Pop}(S)$.

- Se implementada com lista encadeada então
  - a estrutura usa memória nos apontadores, além dos dados propriamente ditos.
  - pode crescer suavemente.
  - get é $O(n)$.
- Se implementadas com vetor
  - não há os apontadores, mas pode haver uma porção do vetor não utilizada.
  - Pode ser necessário redimensionar.
  - get é $O(1)$.

Fila (queue)

- Uma fila armazena uma seqüência de dados onde o primeiro a entrar é o primeiro a sair (FIFO).
- As operações típicas são:
  - get-head.
  - eject.
  - push.

Pilha (stack)

- Uma pilha armazena uma seqüência de dados onde o último a entrar é o primeiro a sair (LIFO).
- As operações típicas são:
  - get-tail.
  - push.
  - pop.

Deque (double ended queue)

- É uma seqüência de dados que permite operações nas duas extremidades.
  - get-head.
  - get-tail.
  - inject.
  - eject.
  - push.
  - pop.

Recursão

Recursão

- Aumentar a solução para um subproblema menor é uma forma de resolver problemas.
- Outra forma de resolver um problemas é combinando soluções para subproblemas menores.
- Há várias estruturas de dados que são recursivas: toda subestrutura tem as mesmas propriedades da estrutura original.

- Recursão
  - é uma forma de pensar no problema.
  - é uma forma de implementar a solução.
- Nem sempre essa forma vai resultar em uma solução eficiente.

Exemplo: o problema da celebridade

- Uma celebridade é uma pessoa que é conhecida por todos mas que não conhece ninguém.
- O problema é determinar se existe uma celebridade em um conjunto de n pessoas fazendo perguntas da forma "Você conhece aquela pessoa?", supondo que todos respondem e ninguém mente.

Solução direta

- A solução direta é perguntar para cada pessoa sobre todas as demais.
- A solução direta faz até $n(n-1)$ perguntas.

Aumentar a solução

- Vamos escolher duas pessoas i e j. Com certeza uma delas não é celebridade. Basta uma pergunta para determinar qual delas. Suponha que seja i.

Aumentar a solução

- Vamos escolher duas pessoas i e j. Com certeza uma delas não é celebridade. Basta uma pergunta para determinar qual delas. Suponha que seja i.
- Suponha que sabemos resolver o problema sem a pessoa i.

Aumentar a solução

- Vamos escolher duas pessoas i e j. Com certeza uma delas não é celebridade. Basta uma pergunta para determinar qual delas. Suponha que seja i.
- Suponha que sabemos resolver o problema sem a pessoa i.
- Resolvido esse problema de tamanho $n-1$, ou não há celebridade nesse conjunto ou a celebridade é uma pessoa k.

Aumentar a solução

- Vamos escolher duas pessoas i e j. Com certeza uma delas não é celebridade. Basta uma pergunta para determinar qual delas. Suponha que seja i.
- Suponha que sabemos resolver o problema sem a pessoa i.
- Resolvido esse problema de tamanho $n - 1$, ou não há celebridade nesse conjunto ou a celebridade é uma pessoa k.
- Vamos aumentar a solução incluindo a pessoa i. Como i não é uma celebridade, então
  - se não havia celebridade dentre as $n - 1$ pessoas não é preciso fazer nada
  - se i não conhece k ou k conhece i então k não é celebridade. Senão k é celebridade.

Algoritmo

CELEBRIDADE(S)

if $|S| == 1$
    $k = 1$
else
    Sejam i, j quaisquer duas pessoas em S
    if i conhece j
        $nc = i$
    else
        $nc = j$
    $S' = S \setminus \{nc\}$
    $k = \text{CELEBRIDADE}(S')$
    if $k > 0$ and (k conhece nc or nc não conhece k)
        $k = 0$
return k

Solução recursiva

- Fazemos até $3(n-1)$ perguntas.

Exemplo: merge-sort

- O problema é ordenar um vetor A com n elementos.
- Vamos supor que sabemos ordenar vetores com $n/2$ elementos.
- Para ordenar A, basta dividir A em duas metades, ordená-las e intercalá-las para obter um vetor ordenado.

Exemplo: merge-sort

$\textsc{Merge-Sort}(A, l, r)$

if $l < r$

$\quad m = \lfloor (l + r)/2 \rfloor$

$\quad \textsc{Merge-Sort}(A, l, m)$

$\quad \textsc{Merge-Sort}(A, m + 1, r)$

$\quad \textsc{Merge}(A, l, m, r)$

- $\textsc{Merge-Sort}(A, l, r)$ ordena as chaves em A no intervalo $[l, r]$.
- $\textsc{Merge}$ é a função que intercala os subvetores ordenados $A[l \,..\, m]$ e $A[m + 1 \,..\, r]$.
- O $\textsc{Merge-Sort}$ é um algoritmo eficiente para ordenação.

Exemplo: listas

- Se removermos o primeiro elemento de uma lista com n nós ficamos com uma lista com $n - 1$ nós.
- É fácil resolver problemas em uma lista com apenas um nó ou em uma lista vazia.

Calcular o tamanho de uma lista

```c
typedef struct node {

  int data;
  struct node* next;

} node;


int length_rec(node* p) {

  if (!p)
    return 0;

  return (1 + length_rec(p->next));
}
```

Copiar uma lista

```
void copy_rec(node* list, node** copy) {

  if (list == NULL)
    *copy = NULL;
  else {
    copy_rec(list->next, copy);

    node* p = (node*) malloc(sizeof(node));
    p->data = list->data;

    p->next = *copy;
    *copy = p;
  }
}
```

Função recursiva

- Um função recursiva é uma função que inclui uma chamada a si mesma em seu corpo.
- Pode haver recursão indireta (ou recursão mútua) quando uma função F chama uma função G, que por sua vez chama F.

- Exemplos tradicionais de recursão são as funções de Fibonacci e a função fatorial.

```
int fib(int n) {
  if (n == 0)
    return 1;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

```
int fat(int n) {
  if (n == 1)
    return 1;
  else
    return n * fat(n-1);
}
```

- Um exemplo tradicional de recursão mútua são as $f(n) = n \bmod 2$ e $g(n) = (n+1) \bmod 2$.

$$f(n) = \begin{cases} 0 & \text{se } n = 0 \\ g(n-1) & \text{se } n > 0 \end{cases}$$

$$g(n) = \begin{cases} 1 & \text{se } n = 0 \\ f(n-1) & \text{se } n > 0 \end{cases}$$

Registros de ativação

- Durante a execução de uma função recursiva, um registro de ativação para cada chamada da função é mantido na stack.
- Um registro de ativação mantém a informação necessária para executar a função (parâmetros e variáveis locais), para retornar o valor dela e para devolver o controle da execução para a função chamadora.
- Essa memória é parte da memória usada pelo programa.

			fat(1)			
		fat(2)	fat(2)	fat(2)		
	fat(3)	fat(3)	fat(3)	fat(3)	fat(3)	
fat(4)	fat(4)	fat(4)	fat(4)	fat(4)	fat(4)	fat(4)

Recursão de cauda

- Seja F uma função que chama G (que pode ser igual a F). A chamada para G é uma chamada de cauda se F retorna o valor retornado por G sem realizar computação adicional.

Recursão de cauda

- Seja F uma função que chama G (que pode ser igual a F). A chamada para G é uma chamada de cauda se F retorna o valor retornado por G sem realizar computação adicional.
- Uma função recursiva F é uma recursão de cauda se todas as chamadas recursivas em F são chamadas de cauda.
- A vantagem da recursão de cauda é que ela pode ser executada usando apenas um registro de ativação, e portanto memória constante.

Outra fatorial

- Uma outra forma de implementar a função fatorial é

```
int fatrc(int n, int res) {
  if (n == 1)
    return res;
  else
    return fatrc(n-1, n * res);
}
```

- `fatrc(n,1)` retorna $n!$.

- `fatrc(n,1)` produz $n-1$ chamadas recursivas, assim como `fat(n)`:

```
fatrc(n-1,n*1),
fatrc(n-2,(n-1)*n*1),
...,
fatrc(1,2*...*(n-1)*n*1).
```

- É fácil ver que o valor retornado por `fatrc(n,res)` é exatamente o mesmo que o valor retornado por `fatrc(n-1,n*res)`.

- `fatrc(n,1)` produz $n-1$ chamadas recursivas, assim como `fat(n)`:

```
fatrc(n-1,n*1),
fatrc(n-2,(n-1)*n*1),
...,
fatrc(1,2*...*(n-1)*n*1).
```

- É fácil ver que o valor retornado por `fatrc(n,res)` é exatamente o mesmo que o valor retornado por `fatrc(n-1,n*res)`.
- Também é fácil ver que o valor retornado pela chamada `fatrc(n,1)` será o mesmo que o retornado pela última chamada recursiva, `fatrc(1,2*...*(n-1)*n*1)`.

- Então não há necessidade realizar todos os retornos desde `fatrc(1,2*...*(n-1)*n*1)` até `fatrc(n,1)`.
- E não há necessidade de armazenar todos os registros de ativação para as chamadas recursivas, basta o registro de ativação para `fatrc(n,1)`.
- Então `fatrc` pode ser executada usando memória constante, independentemente do número de chamadas recursivas que forem feitas.

Conversão

- Uma forma genérica de transformar uma função recursiva em recursão de cauda é:
  - Todo o trabalho realizado após a chamada da função é antecipado.
  - Se não for possível (por causa de dependências) então os valores são passados para a função através de parâmetros adicionais e o trabalho é realizado ao longo das chamadas.
- Os compiladores (quase sempre) fazem esse trabalho de otimização.

- A função Fibonacci com recursão de cauda pode ser construída assim:

```
int fibrc(int n, int t1, int t2) {

  if (n == 0)
    return t1;
  else if (n == 1)
    return t2;
  else
    return fibrc(n-1,t2,t1+t2);
}
```

- A chamada `fibrc(n,1,1)` retorna o n-ésimo número da série.
- Os parâmetros adicionas podem ser escondidos por uma função auxiliar:

```
int fib(int n) {
  return fibrc(n,1,1);
}
```

Eliminação de recursão

- A partir de uma função recursiva sempre é possível escrever uma função equivalente sem recursão.
- A idéia geral é usar uma pilha para simular a stack e os registros de ativação.

```
FIB-SEQUENCE(n)
Let S be an empty stack
PUSH(S, 1)
PUSH(S, 1)
n = n − 2
while n > 0
    x = POP(S)
    y = POP(S)
    z = x + y
    PUSH(S, x)
    PUSH(S, y)
    PUSH(S, z)
    n = n − 1
```

- Um esquema interessante e geral de eliminação de recursão aparece na apostila
  C.L. Lucchesi e T. Kowaltowski. Estruturas de dados e técnicas de programação, 2004.

Árvores enraizadas

Árvore enraizada

- Um árvore enraizada é uma forma encadeada de organizar a memória:
  - Cada nó aponta para zero ou mais filhos.
  - A raiz da árvore não é filha de nenhum nó.
- Várias estruturas de dados são organizadas como uma árvore enraizada ou como uma árvore enraizada binária.

Árvores binárias

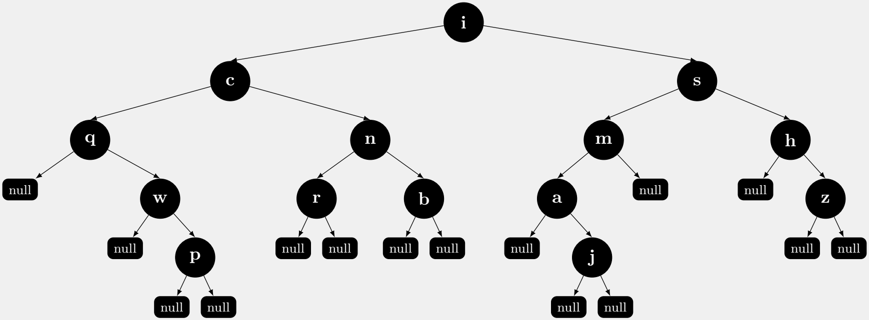
Árvore binária enraizada

- Uma árvore binária enraizada, chamada simplesmente de árvore binária, é formada por nós tais que cada nó tem dois filhos.
- A ordem dos filhos é importante: um é o filho da esquerda e outro é o filho da direita.

Árvore binária enraizada

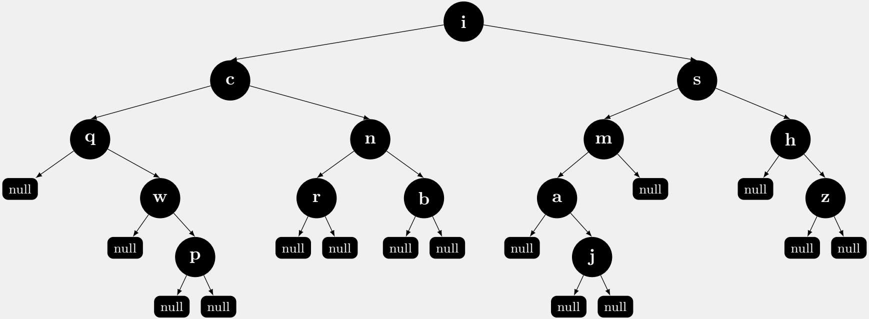
- Uma árvore binária pode ser definida recursivamente da seguinte forma:
    1. Um conjunto vazio de nós é uma árvore binária. A raiz da árvore vazia é nula.
    2. Sejam T_1 e T_2 árvores binárias com raízes r_1 e r_2. Seja r um novo nó. Se r_1 e a r_2 se tornarem filhos de r temos uma árvore binária T com raiz r.

Nomenclatura



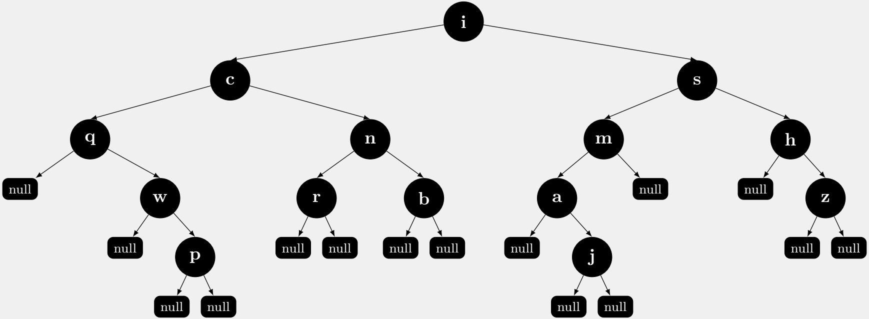
- Uma **aresta orientada** é a ligação entre **pai** e **filho**.
- Dois nós com o mesmo pai são **irmãos**.
- A **raiz** da árvore é um nó sem pai.
- Uma **folha** ou **nó-externo** é um nó sem nós como filhos.
- Nós que têm pelo menos um filho são **nós-internos**.

Nomenclatura



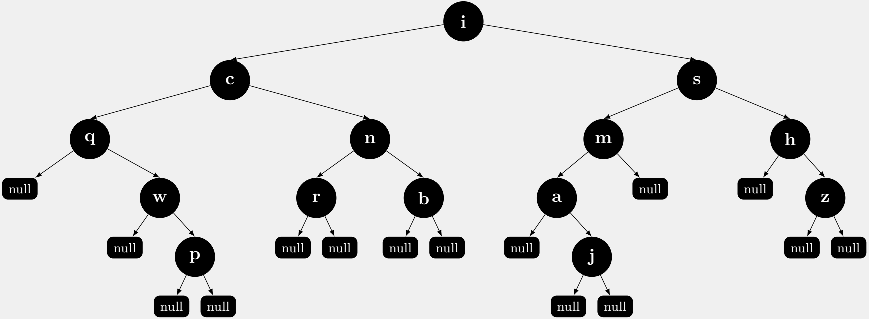
- Um **caminho** é uma sucessão orientada de zero ou mais arestas consecutivas na árvore.
- O **tamanho de um caminho** é o número de arestas que o compõe.

Nomenclatura



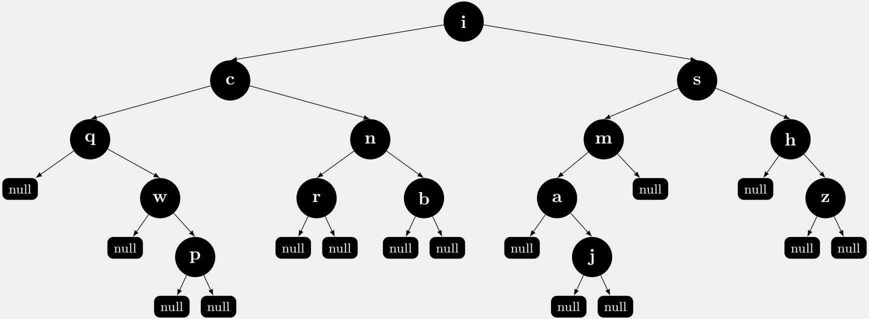
- Um nó u é **ancestral** de um nó v se u está no caminho da raiz até v.
- Um nó u é **descendente** de v se v está no caminho da raiz até u.
- Um nó u e todos os nós que descendem dele formam uma **sub-árvore**.

Nomenclatura



- A **profundidade** de um nó u é o tamanho do caminho da raiz até u.
- Nós com a mesma profundidade estão no mesmo **nível**.
- A raiz está no nível 0.

Nomenclatura



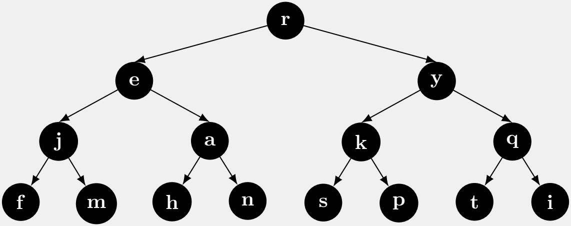
- A **altura** do nó u é o tamanho do maior caminho de u até uma folha.
- Todas as folhas têm altura igual a 0.
- A **altura da árvore** é a altura da raiz.

Fatos

- Uma árvore binária não-nula com n nós tem:
  1. altura máxima $n - 1$ e altura mínima $\lfloor \log_2 n \rfloor$.
  2. n sub-árvores não-vazias e $n + 1$ sub-árvores vazias.
- Uma árvore binária não-nula com altura h tem:
  1. no mínimo $h + 1$ nós e no máximo $2^{h+1} - 1$ nós.
- O caminho da raiz até qualquer nó é único.

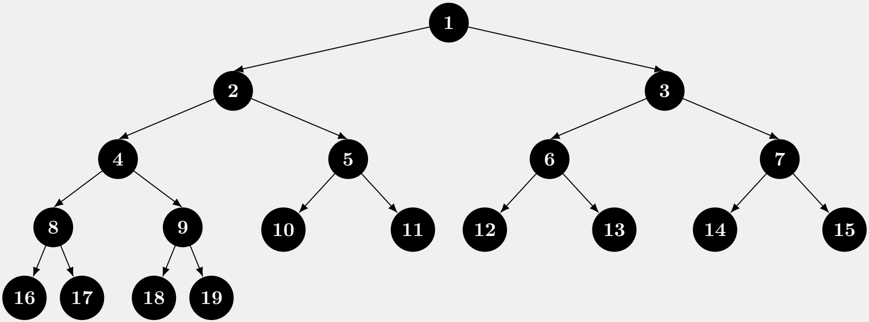
Árvore completa

- Uma árvore binária é **completa** se todos os nós internos têm dois filhos e todas as folhas estão no mesmo nível.



Árvore quase-completa

- Uma árvore binária é **quase-completa** se todos os seus níveis estão preenchidos, exceto talvez pelas folhas à direita do último nível.



Nível de um nó na árvore quase-completa

- Cada nível l de uma árvore quase-completa, exceto talvez pelo último, tem exatamente 2^l nós.
- Os nós em um nível l podem ser rotulados

$$2^l, 2^l + 1, 2^l + 2, \ldots, 2^{l+1} - 1.$$

Se o nó i está no nível l então

$$\begin{aligned} 2^l \leq & \quad i & < 2^{l+1} \\ \lg 2^l \leq & \quad \lg i & < \lg 2^{l+1} \\ l \leq & \quad \lg i & < l + 1 \end{aligned}$$

e então $l = \lfloor \lg i \rfloor$.

Altura de um nó na árvore quase-completa

- A altura h do nó i é o tamanho do maior caminho de i até alguma folha. Tomando sempre o filho da esquerda temos o caminho de tamanho máximo

  $$i, 2i, 2^2 i, \ldots, 2^h i$$

  para $2^h i \leq n < 2^{h+1} i$.

  Então

  $$\begin{array}{rcl} 2^h i \leq & n & < 2^{h+1} i \\ 2^h \leq & \frac{n}{i} & < 2^{h+1} \\ \lg 2^h \leq & \lg \frac{n}{i} & < \lg 2^{h+1} \\ h \leq & \lg \frac{n}{i} & < h+1 \end{array}$$

  e $h = \lfloor \lg \frac{n}{i} \rfloor$.

Percursos em uma árvore

- Em uma árvore há várias formas de percorrer os nós a partir da raiz.
- Alguns percursos são úteis para obter informações a respeito da árvore ou dos dados armazenados na árvore.
- Há dois principais:
  1. em profundidade: "para baixo primeiro".
  2. em largura: "para o lado primeiro".
- Durante o percurso geralmente realiza-se alguma operação em cada nó, que vamos chamar genericamente de **visitar**.

Percursos em profundidade

- São três, definidos recursivamente. O nome indica a visitação da raiz relativamente às sub-árvores dela:

Percursos em profundidade

- São três, definidos recursivamente. O nome indica a visitação da raiz relativamente às sub-árvores dela:
  - pré-ordem:
    1. Visitar a raiz
    2. Percorrer a sub-árvore esquerda
    3. Percorrer a sub-árvore direita

Percursos em profundidade

- São três, definidos recursivamente. O nome indica a visitação da raiz relativamente às sub-árvores dela:
  - pré-ordem:
    1. Visitar a raiz
    2. Percorrer a sub-árvore esquerda
    3. Percorrer a sub-árvore direita
  - em-ordem:
    1. Percorrer a sub-árvore esquerda
    2. Visitar a raiz
    3. Percorrer a sub-árvore direita

Percursos em profundidade

- São três, definidos recursivamente. O nome indica a visitação da raiz relativamente às sub-árvores dela:
  - pré-ordem:
    1. Visitar a raiz
    2. Percorrer a sub-árvore esquerda
    3. Percorrer a sub-árvore direita
  - em-ordem:
    1. Percorrer a sub-árvore esquerda
    2. Visitar a raiz
    3. Percorrer a sub-árvore direita
  - pós-ordem:
    1. Percorrer a sub-árvore esquerda
    2. Percorrer a sub-árvore direita
    3. Visitar a raiz

Pré-ordem

PRE-ORDER(x)

if $x \neq$ NULL
    visit x
    PRE-ORDER($x.\mathit{left}$)
    PRE-ORDER($x.\mathit{right}$)

Pré-ordem iterativa

```
PRE-ORDER(x)
Let S be an empty stack
PUSH(S, x)
while S ≠ ∅
    x = POP(S)
    if x ≠ NULL
        visit x
        PUSH(S, x.right)
        PUSH(S, x.left)
```

Em-ordem

```
IN-ORDER(x)
if x ≠ NULL
        IN-ORDER(x.left)
        visit x
        IN-ORDER(x.right)
```

Em-ordem iterativa

```
IN-ORDER(x)
Let S be an empty stack
while IS-NOT-EMPTY(S) or x ≠ NULL
    if x ≠ NULL
        PUSH(S, x)
        x = x.left
    else
        x = POP(S)
        visit x
        x = x.right
```

Pós-ordem

```
POST-ORDER(x)
if x ≠ NULL
    POST-ORDER(x.left)
    POST-ORDER(x.right)
    visit x
```

Pós-ordem iterativa

```
POST-ORDER(x)
Let S be an empty stack
while IS-NOT-EMPTY(S)
    while x ≠ NULL
        if x.right ≠ NULL
            PUSH(S, x.right)
        PUSH(S, x)
        x = x.left
    x = POP(S)
    if x.right ≠ NULL and x.right == AT_TOP(S)
        y = POP(S)
        PUSH(S, x)
        x = x.right
    else
        visit x
        x = NULL
```

Percurso em largura

BREADTH($root$)

Let Q be an empty queue
ENQUEUE($root$)
while $Q \neq \emptyset$
    node p = DEQUEUE(Q)
    visit p
    if $p.\mathit{left} \neq$ NULL
        ENQUEUE($Q, p.\mathit{left}$)
    if $p.\mathit{right} \neq$ NULL
        ENQUEUE($Q, p.\mathit{right}$)

Representação explícita ou encadeada

- Cada nó tem a chave e um apontador para o filho da esquerda e um apontador para o filho da direita.
- Vamos chamar os apontadores no nó v de $v.left$ e $v.right$.

Representação com apontador para o pai

- Usa três apontadores em cada nó: um para o pai e dois para os filhos.
- Permite percorrer a árvore das folhas para a raiz.
- Permite realizar percursos sem usar uma pilha.
- Usa mais memória.

Representação costurada (threaded)

- As folhas têm dois apontadores nulos. São mais numerosos que os demais. (É muita memória apontando para nada.)

Representação costurada (threaded)

- As folhas têm dois apontadores nulos. São mais numerosos que os demais. (É muita memória apontando para nada.)
- Os apontadores nulos nas folhas podem ser redefinidos assim:
  - o filho da esquerda aponta para o predecessor em-ordem,
  - o filho da direita aponta para o sucessor em-ordem e
  - os extremos na ordem apontam para a raiz.

Representação costurada (threaded)

- As folhas têm dois apontadores nulos. São mais numerosos que os demais. (É muita memória apontando para nada.)
- Os apontadores nulos nas folhas podem ser redefinidos assim:
  - o filho da esquerda aponta para o predecessor em-ordem,
  - o filho da direita aponta para o sucessor em-ordem e
  - os extremos na ordem apontam para a raiz.
- Dessa forma é possível percorrer os nós em-ordem sem usar recursão e sem usar uma pilha.

Representação costurada (threaded)

- As folhas têm dois apontadores nulos. São mais numerosos que os demais. (É muita memória apontando para nada.)
- Os apontadores nulos nas folhas podem ser redefinidos assim:
  - o filho da esquerda aponta para o predecessor em-ordem,
  - o filho da direita aponta para o sucessor em-ordem e
  - os extremos na ordem apontam para a raiz.
- Dessa forma é possível percorrer os nós em-ordem sem usar recursão e sem usar uma pilha.
- O nó precisa registrar se cada apontador é para um filho ou é threaded. Um bit por apontador é suficiente.

- Os percursos em-ordem e em-ordem-inversa ficam muito eficientes. O custo adicional para manter os threads é baixo.
- Pode ser costurada apenas nos apontadores direitos ou apenas nos esquerdos, se apenas uma das ordens for necessária.

Representação implícita ou seqüencial

- Os nós de uma árvore podem ser colocados em um vetor de tal forma que
  1. a raiz está na posição 0 e
  2. os filhos do nó i estão nas posições $2i + 1$ e $2i + 2$.
- Dessa forma, o pai do nó i está em $\lfloor \frac{i-1}{2} \rfloor$.
- Se a árvore é quase-completa então essa representação usa pouca memória.

Representação como vetor de predecessores

- Os nós de uma árvore podem ser colocados em um vetor P de tal forma que
  1. $P[i]$ é o índice do pai do nó i.
  2. $P[raiz]$ é o índice da própria raiz.
- Para percorrer um caminho da raiz até um nó i percorremos um caminho de i até a raiz e empilhamos os nós ao longo do caminho.
- Essa representação usa pouca memória mesmo se a árvore não é quase-completa, mas encontrar os nós e percorrê-la leva mais tempo.

```
PRINT-PATH(P, i)
if i == P[i]
    print i
else
    PRINT-PATH(P, P[i])
    print i
```