Algoritmos e Estruturas de Dados 2

Árvores

Sedgewick (Cap.5: 5.4-5.7); Cormen (Sec. 10.4)

Prof. Fábio M. Costa

Instituto de Informática
Universidade Federal de Goiás

1o. Semestre / 2019

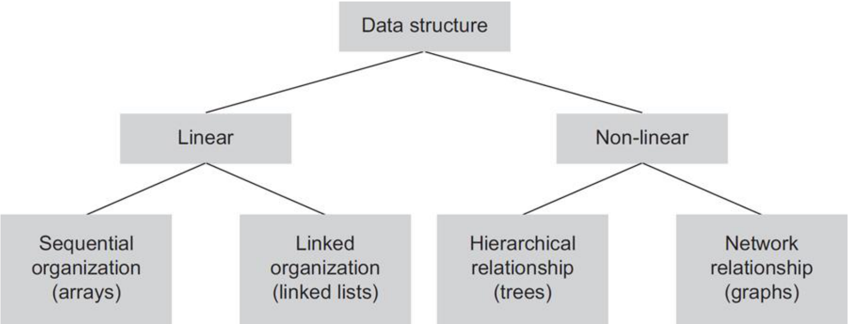
Aplicações de Árvores

Em geral: Manipulação de dados hierárquicos, tornar a busca mais eficiente, manipulação de coleções ordenadas de dados (geralmente dinâmicas)

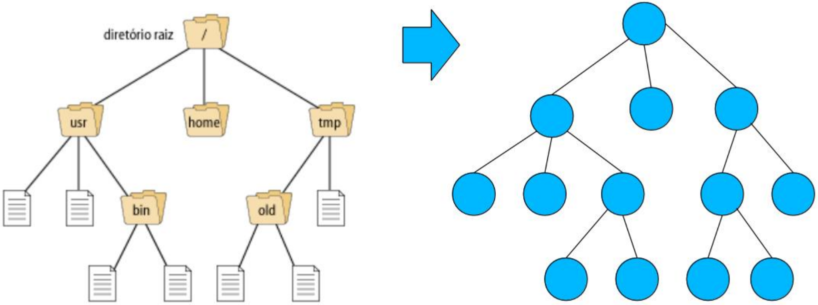
Exemplos

- Melhorar o tempo de busca em bases de dados (árvores de busca binárias, árvores AVL, árvores rubro-negras)
- Programação de jogos (árvores minmax, árvores de decisão, árvores de busca de caminho)
- Computação gráfica 3D (árvores binárias, quadtrees, octrees)
- Implementação de linguagens de programação (árvores de sintaxe abstrata, árvores de precedência de expressões aritméticas)
- Compressão de dados (árvores de Huffman)
- Sistemas de arquivos (árvores B, árvores indexadas esparsas, árvores trie)

Estruturas de Dados Lineares e Não-Lineares



Exemplo: Sistema de Arquivos



...

Terminologia Básica: Grafos

Árvores são um caso restrito de grafos

Grafo: G = (V, E)

- V = conjunto de vértices (ou nós)
- E = conjunto de arestas
- Mapeamento de E para V

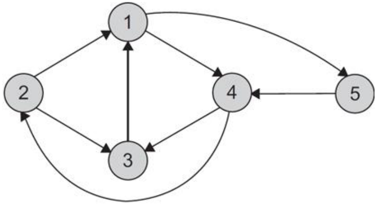
Nós $a, b \in V$ são adjacentes se

Existe uma aresta $\ell \in E$ que associa a e b

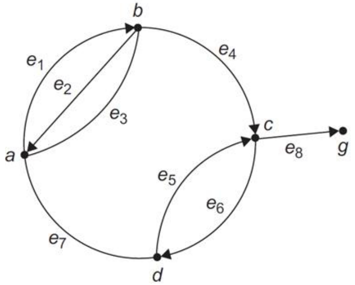
Grafos direcionados e não-direcionados

- Em um grafo, as arestas podem ou não ter uma direção específica
- Grafo direcionado: todas as arestas tem uma direção especificada
- Grafo não-direcionado: nenhuma aresta tem direção específica
- Grafo misto: possui arestas direcionadas e não-direcionadas

Arestas Paralelas e Multigrafos



(a)



(b)

- Duas arestas incidentes no mesmo par de nós, mas com direções opostas são consideradas distintas
- **Arestas paralelas:** incidem no mesmo par de nós
- **Multigrafo:** grafo que contém arestas paralelas
- **Grafo simples:** sem arestas paralelas

Grafos: definições básicas

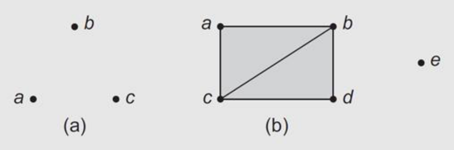
Grafos Ponderados

Pesos são associados às arestas e/ou aos vértices do grafo
Exemplo: mapa da cidade modelado como grafo

- cruzamentos: vértices
- segmentos de ruas: areastas
- intensidade do tráfego: peso das arestas

Grafo nulo e Vértice isolado

Nó ou vértice isolado: não tem nós adjacentes
Grafo nulo: contém apenas vértices isolados (e o conjunto de arestas do grafo é vazio)



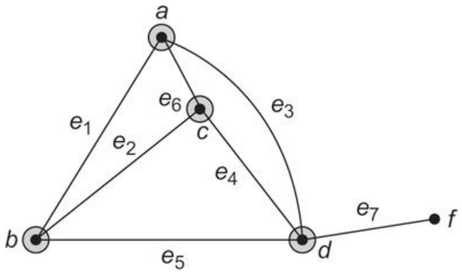
Grafos: Grau de um Vértice

Em grafos direcionados

Grau sainte (*outdegree*) de um vértice V**:** número de arestas que têm V como origem
Grau entrante (*indegree*): número de arestas que têm V como destino
Grau total: soma do *outdegree* com o *indegree*) de V

Em grafos não-direcionados

Grau de um nó V**:** número de arestas incidentes em V



Grafos: Caminhos e Circuitos (em grafos simples)

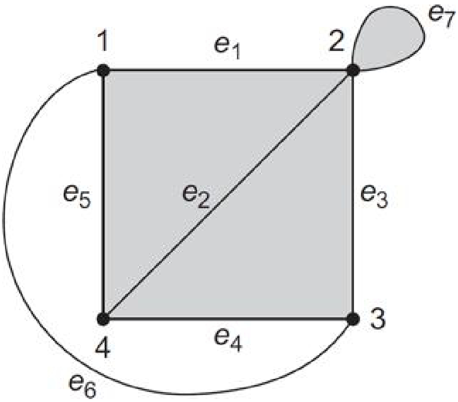
Caminho:

Sequência de arestas tal que o nó terminal de uma aresta é o nó inicial da próxima aresta na sequência.

Caminho simples: nenhuma aresta é visitada mais de uma vez
Caminho elementar: nenhum nó é visitado mais de uma vez

Circuito (ciclo)

Um caminho que origina e termina no mesmo nó.
Ciclo simples
Ciclo elementar

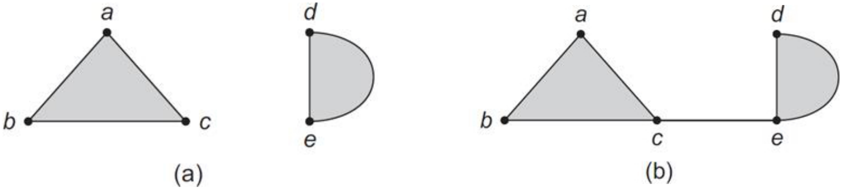


Grafos: Conectividade

Grafo Conectado

Se há um caminho entre todo e qualquer par de nós

Exemplos:



Grafos e Árvores

Grafo Acíclico

Grafo simples sem ciclos (sem loops)

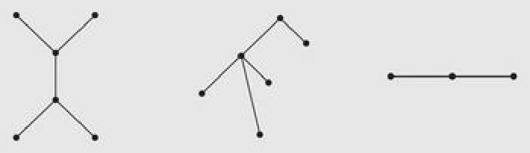
Árvores

Uma classe de grafos acíclicos

Florestas e Árvores

- **Floresta:** Um grafo que não contém ciclos
- **Árvore:** Uma floresta conectada (e sem ciclos)
- Árvores e florestas são grafos simples

Exemplo:



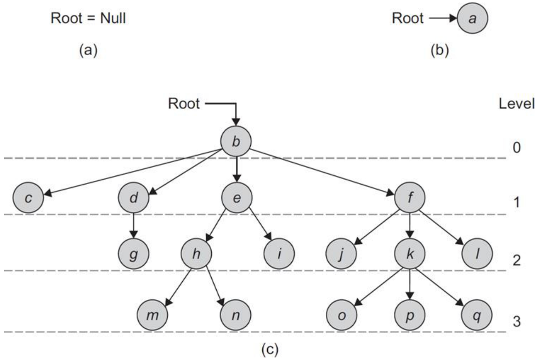
Árvores: Definições básicas

Em árvores direcionadas...

- **Árvore Direcionada:** um grafo acíclico direcionado
- **Raiz:** único nó com grau entrante (*indegree*) igual a zero
- **Nó Folha (ou terminal):** qualquer nó com grau sainte (*outdegree*) igual a zero
- **Nó interno:** qualquer outro nó cujo grau sainte (*outdegree*) é diferente de zero
- **Nível de um nó:** comprimento (em número de arestas) do caminho da raiz até o nó; o nível da raiz é zero.

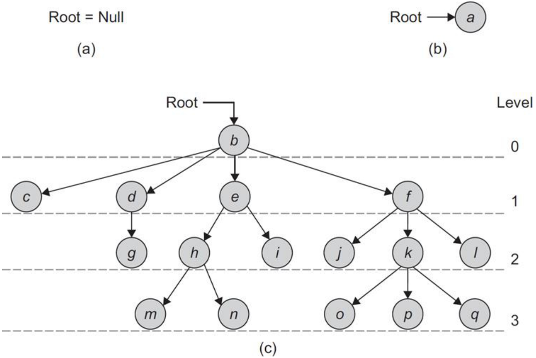
Árvores: Definição Geral

1. Um conjunto de nós tal que:
   - existe um nó R, denominado raiz, com zero ou mais sub-árvores, cujas raízes estão ligadas a R
   - os nós raiz dessas sub-árvores são os filhos de R
2. Um conjunto de zero nós é uma árvore vazia



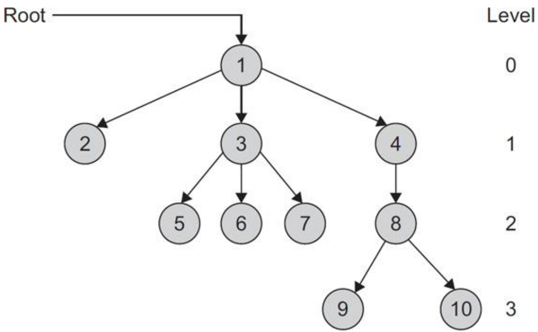
Grau de uma Árvore

- **Grau de um nó:** número de sub-árvores que o nó possui
- **Grau de uma árvore:** grau máximo dentre os nós da árvore
- O grau de uma árvore vazia é indefinido



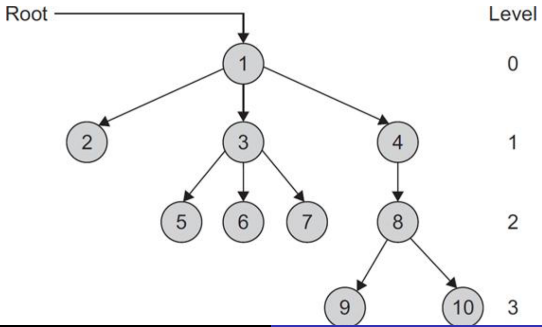
Algumas observações

- A relação entre um nó pai e seus nós filhos (raízes de suas sub-árvores) implica na direcionalidade das arestas:
  - uma aresta inicia no nó pai e termina no nó filho
- **Nível de um nó:** comprimento do caminho da raiz até o nó
- **Altura de uma árvore:** comprimento do caminho da raiz até o nó de mais baixo nível (i.e., comprimento do maior caminho na árvore)



Percurso em Árvores

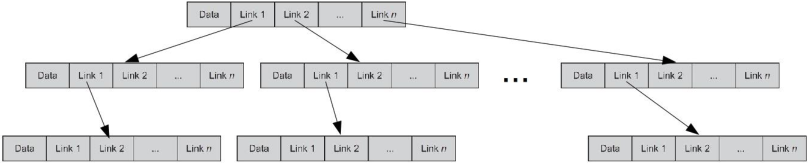
- **Pré-Ordem:** visita a raiz e, em seguida, visita, em pré-ordem, os nós das sub-árvores da raiz da esquerda para a direita.
- **In-Ordem:** visita, em in-ordem, a sub-árvore mais à esquerda; então visita a raiz e, em seguida, visita, em in-ordem, as demais sub-árvores da esquerda para a direita.
- **pós-ordem:** visita, em pós-ordem, as sub-árvores da raiz da esquerda para a direita; então visita a raiz.



Representação Genérica de Árvores (1)

Nós com estrutura fixa

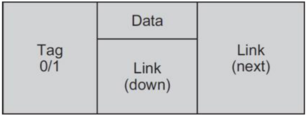
Cada nó consiste em um campo de dados e um vetor de ponteiros



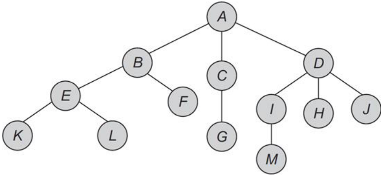
Representação Genérica de Árvores (2)

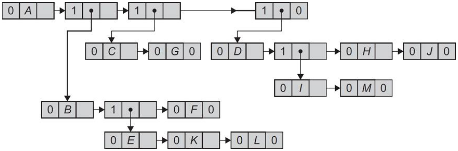
Nós com estrutura variável

Cada nó (da árvore) consiste em uma lista ligada de nós com a seguinte estrutura:



Exemplo:



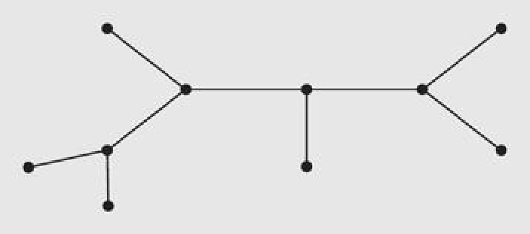


Tipos de Árvores (quanto à topologia

- Árvore livre
- Árvore enraizada
- Árvore ordenada
- Árvore regular
- Árvore binária
- Árvore completa
- Árvore de posição

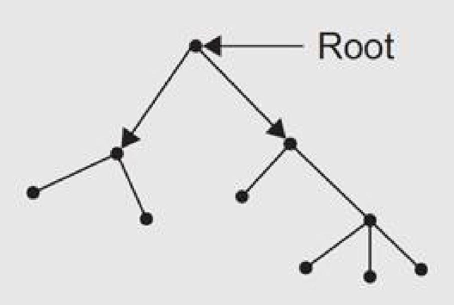
Árvore Livre

- Um grafo conectado, acíclico e não direcionado.
- Nenhum nó é designado como raiz



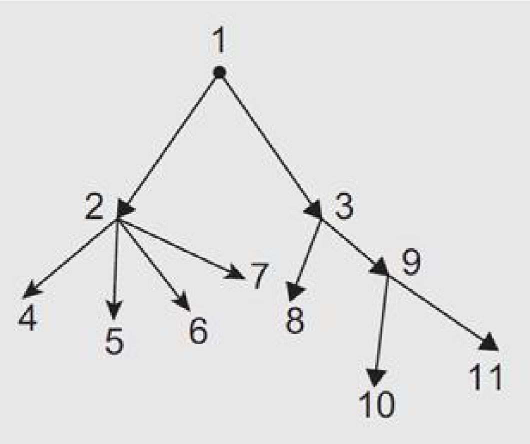
Árvore Enraizada (Rooted Tree)

- Um grafo direcionado onde um nó é designado para ser a raiz
- O grau entrante da raiz é zero
- O grau entrante dos demais nós é 1



Árvore Ordenada

- A ordem relativa dos nós em um certo nível é significativa



Árvore Regular

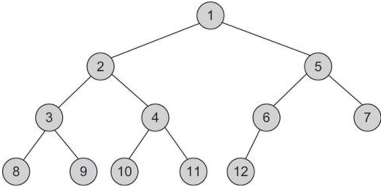
Uma árvore em que todo nó interno tem o mesmo grau sainte (*outdegree*).

- Uma árvore direcionada cujos nós tem *outdegree* menor ou igual a m é chamada de árvore m-ária
- Se o *outdegree* de todos os nós é exatamente m (no caso dos nós não-folha) ou zero (no caso dos nós folha), a árvore é chamada de árvore m-ária regular.

Árvore Binária – Árvore m-ária com $m = 2$

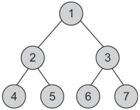
Árvore Binária Completa

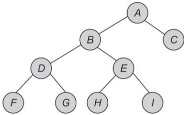
1. Todos os níveis, exceto o último, possuem o número máximo possível de nós
2. Todos os nós do último nível são dispostos à esquerda
3. Em todos os níveis, os nós são preenchidos (ou ordenados) da esquerda para a direita



Árvore Binária Cheia

- Todo nó possui ou 2 ou 0 filhos
  - a.k.a. strictly binary tree
- Caso particular: todos os níveis (incluindo o mais baixo) possuem o número máximo de nós – árvore binária perfeita
  - Todos os nós internos possuem 2 filhos e todas as folhas estão no mesmo nível
  - Número de nós: $2^0 + 2^1 + 2^2 + ... + 2^h = 2^{h+1} - 1$

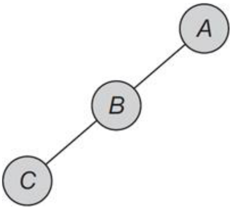




Casos particulares

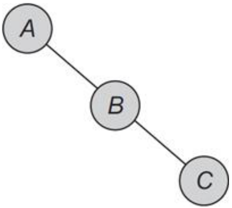
Left-skewed binary tree

Nenhum nó possui sub-árvore da direita



Right-skewed binary tree

Nenhum nó possui sub-árvore da esquerda

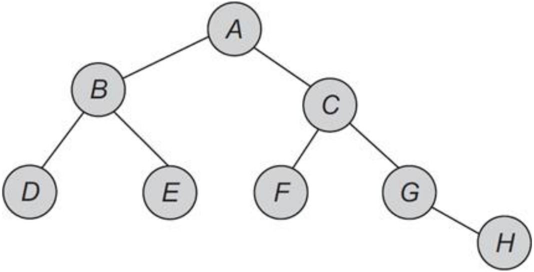


Árvores Binárias

Definição recursiva

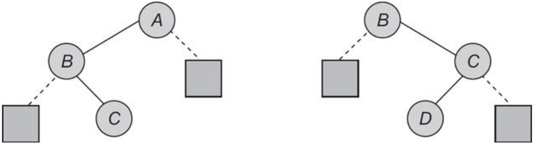
1. É uma árvore vazia, ou
2. Consiste de um nó, a raiz, e dois filhos, um à esquerda e outro à direita, os quais são também árvores binárias.

$\Longrightarrow$ Todos os nós internos de uma árvore binária são raízes de árvores binárias menores.



Árvores Binárias: Consequência da definição

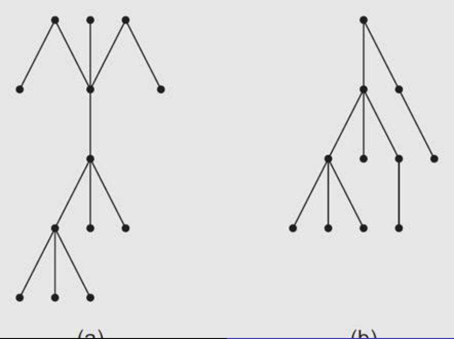
Cada nó não-vazio possui dois filhos, os quais podem ser árvore vazias



Árvores Binárias: Propriedades

Uma árvore é um grafo conectado acíclico

1. Existe um único caminho entre quaisquer dois nós
2. O número de nós é 1 a mais que o número de arestas
3. Uma árvore com 2 ou mais nós tem pelo menos 2 folhas
4. O número máximo de nós de profundidade d em uma árvore binária é 2^d (considerando que a profundidade da raiz é zero)



Tipo Abstrado de Dados – Árvore Binária

```
ADT btree
1.  Declare create() -> btree
2.  makebtree(btree, element, btree) -> btree
3.  isEmpty(btree) -> boolean
4.  leftchild(btree) -> btree
5.  rightchild(btree)-> btree
6.  data(btree) -> element
7.  for all l, r in btree, e in element, Let
8.  isEmpty(create) = true
9.  isEmpty(makebtree(l, e, r)) = false
10. leftchild(create()) = error
11. rightchild(create()) = error
12. leftchild(makebtree(l, e, r)) = l
13. rightchild(makebtree(l, e, r)) = r
14. data(makebtree(l, e, r)) = e
15. end
end btree
```

Implementação (primeiro passo: interface)

```

class TreeNode
{
 public:
  char Data;
  TreeNode *Lchild;
  TreeNode *Rchild;
};

class BinaryTree
{
 private:
  TreeNode *Root;
 public:
  BinaryTree(){Root = Null};
  // constructor creates an empty tree
  TreeNode * GetNode(char);
  void InsertNode(TreeNode*);
  void DeleteNode(TreeNode*);
};
```

Operações em Árvores Binárias

- Creation – Creating an empty binary tree to which the "root" points
- Traversal – Visiting all the nodes in a binary tree
- Deletion – Deleting a node from a non-empty binary tree
- Insertion – Inserting a node into an existing (may be empty) binary tree
- Merge – Merging two binary trees
- Copy – Copying a binary tree
- Compare – Comparing two binary trees
- Finding a replica or mirror of a binary tree

Implementação de Árvores Binárias

Duas alternativas

→ Estrutura ligada (com ponteiros)
→ Estrutura sequencial (array)

Vantagens da estrutura ligada:

- representação natural para árvores binárias
- mais conveniente para inserções e remoções

Implementação de Árvores Binárias usando Arrays

Ideia geral

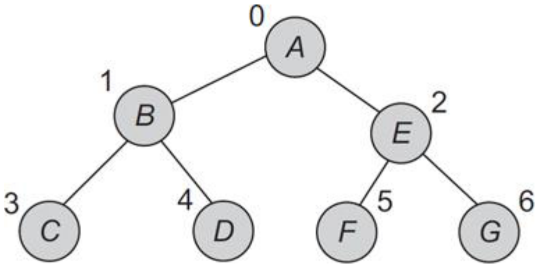
1. Armazenar os nós nível por nível, começando pela raiz (nível 0)
2. Requer numeração sequencial dos nós

Considere uma árvore binária completa de altura h

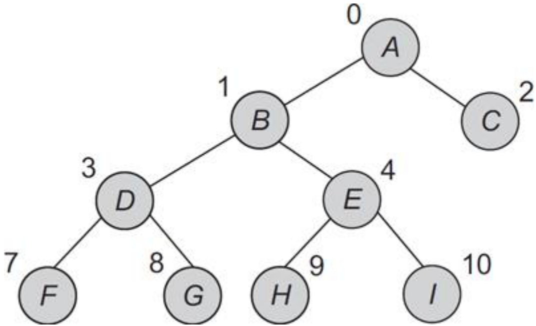
- $2^{h+1} - 1$ nós
- array unidimensional tree[] de comprimento $2^{h+1} - 1$
- nó raiz: na posição tree[0]
- Pai(i) $= \lfloor (i-1)/2 \rfloor$, se $i \neq 0$; do contrário i é a raiz (não tem pai)
- FilhoEsq(i) $= 2i + 1$, se $2i + 1 \leq n - 1$; se $2i \geq n$, não tem filho esq.
- FilhoDir(i) $= 2i + 2$, se $2i + 2 \leq n - 1$; se $2i + 1 \geq n$, sem filho dir.

Implementação de Árvores Binárias usando Arrays: Exemplo

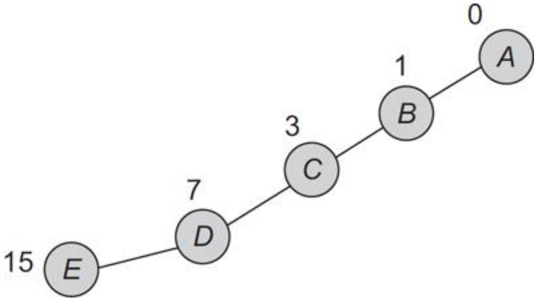
- Pai(i) $= \lfloor (i-1)/2 \rfloor$, se $i \neq 0$; do contrário i é a raiz (não tem pai)
- FilhoEsq(i) $= 2i+1$, se $2i+1 \leq n-1$; se $2i \geq n$, não tem filho esq.
- FilhoDir(i) $= 2i+2$, se $2i+2 \leq n-1$; se $2i+1 \geq n$, sem filho dir.



Implementação de Árvores Binárias usando Arrays: Exemplo



Implementação de Árvores Binárias usando Arrays: Exemplo



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		19
A	*B*	–	*C*	–	–	–	*D*	–	–	–	–	–	–	–	*E*	...	–

Implementação de Árvores Binárias usando Arrays: Considerações

Vantagens

- Muito simples: qualquer nó pode ser acessado a partir de outro nó por meio do cálculo de índices
- Sem o overhead de memória com armazanamento de ponteiros
- Única opção em linguagens sem alocação dinâmica de memória
- Eficiente para árvores binárias completas

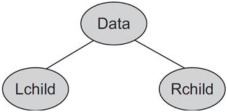
Desvantagens

- Desperdício de memória se árvore não completa
- Árvores desbalanceadas: muitas posições do array ficam ociosas
- Para uma árvore totalmente desbalanceada: array com $2^{k+1} - 1$ elementos. Quantos são ocupados?
- Alocação estática do vetor
- Muitas movimentações de dados para inserções e remoções

Árvores Binárias: Implementação Ligada

Um nó possui três campos:

1. Link (ponteiro) para o filho da esquerda (nulo se folha)
2. Dados
3. Link (ponteiro) para o filho da direita (nulo se folha)



Ponteiro para o nó Pai

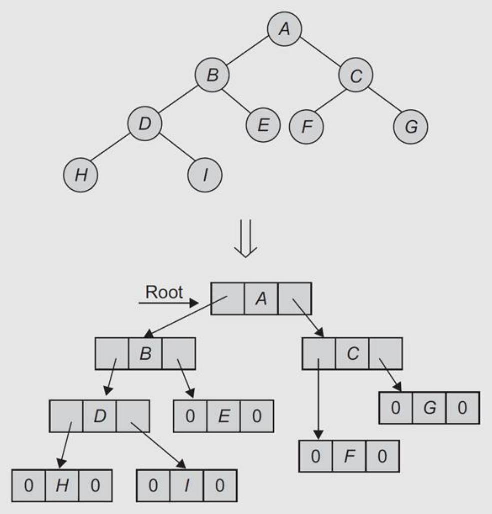
Pode ser necessário para facilitar percursos na árvore

Requer um quarto campo:

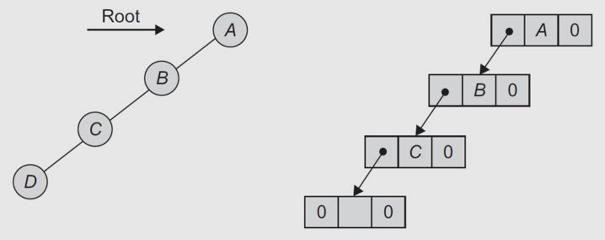
- Link (ponteiro) para o nó pai (nulo se for a raiz)

Árvores Binárias: Implementação Ligada

Exemplo



Árvores Binárias: Implementação Ligada
Exemplo

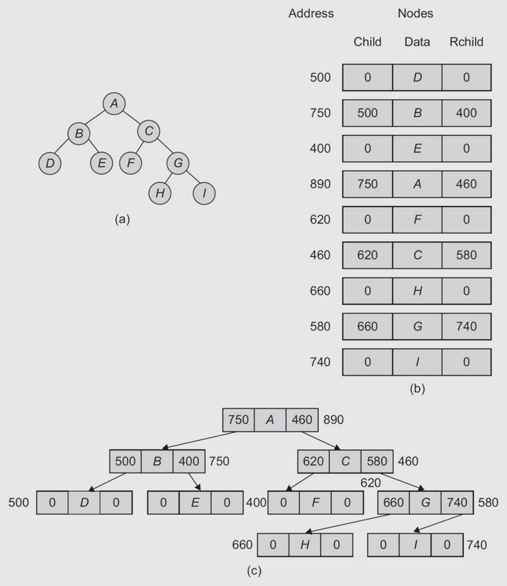


Implementação do TAD Árvore Binária (com ponteiros)

```
class TreeNode
{
 public:
  char Data;
  TreeNode *Lchild;
  TreeNode *Rchild;
};

class BinaryTree
{
 private:
  TreeNode *Root;
 public:
  BinaryTree() {Root = Null;} // constructor
  TreeNode *GetNode(char);
  void InsertNode(TreeNode*);
  void DeleteNode(TreeNode*);
  void Postorder(TreeNode*);
  void Inorder(TreeNode*);
  void Preorder(TreeNode*);
  TreeNode *TreeCopy();
  void Mirror();
  int TreeHeight(TreeNode*);
  int CountLeaf(TreeNode*);
  int CountNode(TreeNode*);
  void BFS_Tree();
  void DFS_Tree();
  TreeNode *Create_Btree_InandPre_Traversal(char preorder[max], char inorder[max]);
  void Postorder_Non_Recursive(void);
  void Inorder_Non_Recursive();
  void Preorder_Non_Recursive();
  int BTree_Equal(BinaryTree, BinaryTree);
  TreeNode *TreeCopy(TreeNode*);
  void Mirror(TreeNode*);
};
```

Alocação dos nós em memória



Árvores Binárias: Implementação Ligada
Considerações

Vantagens

- Não é necessário saber a profundidade da árvore a priori
- Sem desperdício de memória para árvores não-balanceadas
- Inserção e remoção são mais eficientes
- Útil quando o conjunto de dados é dinâmico

Desvantagens

- Acesso direto aos nós não é possível $\rightarrow$ requer percurso a partir da raiz
- Maior quantidade de memória necessária para armazenar cada nó (overhead resultante dos ponteiros)

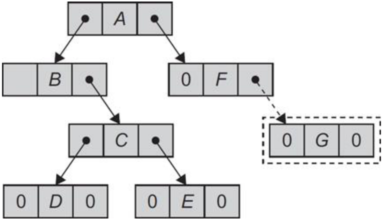
Árvore Binária: Inserção

Em geral, pode ocorrer tanto a inserção de nós internos quanto de nós folha

Caso de uso prático: inserção de nós folha

Inserção de um nó folha

1. Busca pelo nó cujo filho será inserido
2. Faz o nó apontar para o novo filho (direita ou esquerda, dependendo de algum critério)



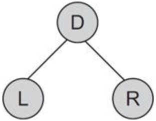
Percurso em Árvores Binárias

Visitar os nós da árvore (cada nó exatamente uma vez)
É uma operação básica: usada como parte de outras operações (busca, lista dos nós, inserção, remoção etc.)
Percurso completo: todos os nós
Operações primitivas de percurso:

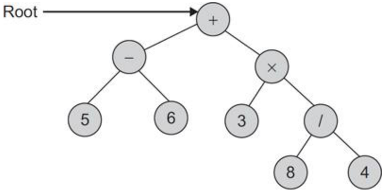
- L: mover p/ filho da esquerda
- R: mover p/ filho da direita
- D: ler os dados do nó

Considerando um nó e suas duas sub-árvores, há seis ordens possíveis de uso dessas primitivas para percorrer uma árvore:

- LDR, LRD, DLR, DRL, RDL e RLD



Percurso em Árvores Binárias
Exemplo



LDR:	5 − 6 + 3 × 8 / 4
LRD:	5 6 − 3 8 4 / × +
DLR:	+ − 5 6 × 3 / 8 4
DRL:	+ × / 4 8 3 − 6 5
RDL:	4 / 8 × 3 + 6 − 5
RLD:	4 8 / 3 × 6 5 − +

Ordens significativas:

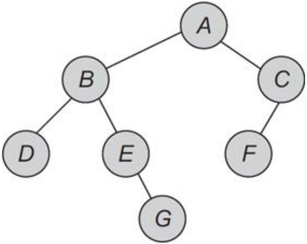
Se adotarmos a convenção de percurso da esquerda p/ a direita:

- LDR, LRD e DLR: inorder, postorder e preorder
- Em expressões aritméticas: infixa, pós-fixada pré-fixada

Percurso em Árvores Binárias: Pré-Ordem

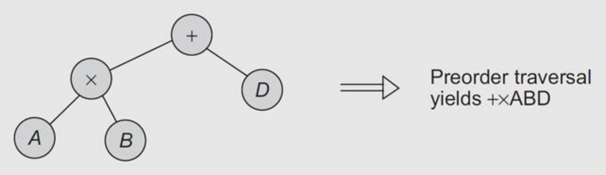
Algoritmo DLR

1. Visita o nó raiz (D)
2. Percorre a sub-árvore da esquerda em pré-ordem (L)
3. Percorre a sub-árvore da direita em pré-ordem (R)



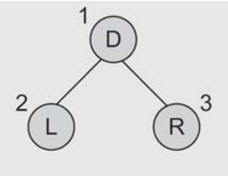
→ Também conhecido como percurso em profundidade.

Percurso em Árvores Binárias: Pré-Ordem
Exemplo: Expressões aritméticas

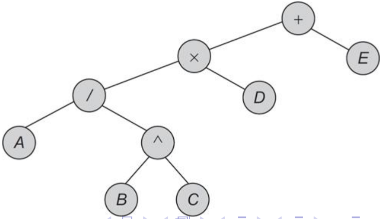


Percurso em Árvores Binárias: Pré-Ordem
Implementação recursiva

```
void BinaryTree :: Preorder(TreeNode*)
{
 if(Root != Null)
 {
  cout << Root->Data;
  Preorder(Root->Lchild);
  Preorder(Root->Rchild);
 }
}
```



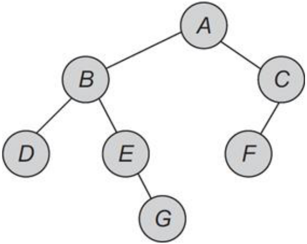
Exemplo:



Percurso em Árvores Binárias: In-ordem

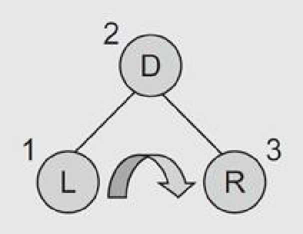
Algoritmo LDR

1. Percorre a sub-árvore da esquerda em in-ordem (L)
2. Visita a raiz (D)
3. Percorre a sub-árvore da direita em in-ordem (R)



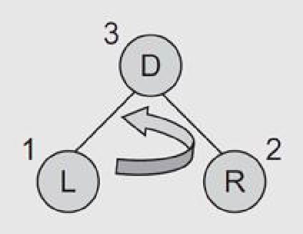
Percurso em Árvores Binárias: In-Ordem
Implementação recursiva

```
void BinaryTree :: Inorder(TreeNode*)
{
 if(Root != Null)
 {
  Inorder(Root?Lchild);
  cout << Root?Data;
  Inorder(Root?Rchild);
 }
}
```



Percurso em Árvores Binárias: Pós-Ordem
Implementação recursiva

```
void BinaryTree :: Postorder(TreeNode*)
{
 if(Root != Null)
 {
  Postorder(Root?Lchild);
  Postorder(Root?Rchild);
  cout << Root?Data;
 }
}
```



Exercício

Implementação não-recursiva dos três percursos em árvores binárias.

Reconstituição de uma Árvore Binária a partir de dois Percursos Conhecidos

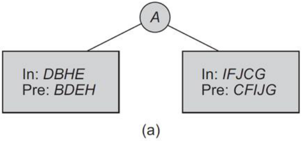
- Não é possível reconstituir uma árvore binária única a partir de um único percurso conhecido.
- É necessário que se conheça dois percursos diferentes:
  - In-ordem e Pré-ordem; ou
  - In-ordem e Pós-ordem.

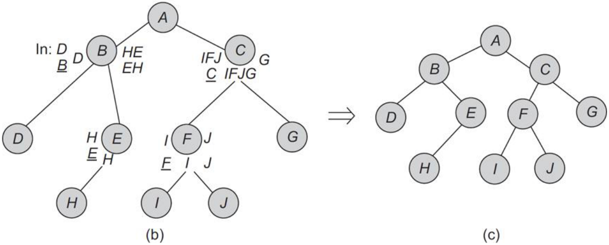
Princípio básico

1. Determinação do nó raiz:
   - Se o percurso em pré-ordem é dado, então o primeiro nó da lista é a raiz
   - Se o percurso em pós-ordem é dado, então o último nó da lista é a raiz
2. Uma vez que a raiz é conhecida, todos os nós em todas as sub-árvores da esquerda e da direita podem ser determinados.
3. A mesma técnica pode ser aplicada repetidamente (recursivamente) para reconstituir as sub-árvores.

Exemplo 1: Construir uma árvore binária a partir de dois percursos conhecidos

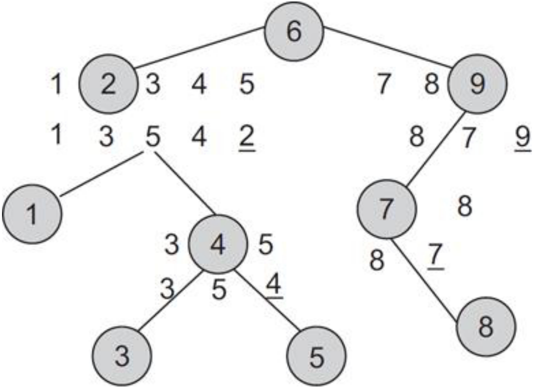
- In-ordem: D B H E A I F J C G
- Pré-ordem: A B D E H C F I J G





Exemplo 2: Construir uma árvore binária a partir de dois percursos conhecidos

- In-ordem: 1 2 3 4 5 6 7 8 9
- Pós-ordem: 1 3 5 4 2 8 7 9 6



Exercício

Implementar a construção de uma árvore binária a partir dos percursos in-ordem e pré-ordem.
Entrada: duas listas de valores to tipo `char` representando os dois percursos.
Saída: uma listagem, em pós-ordem, dos valores dos nós da árvore construída.

Percurso em Largura e em Profundidade

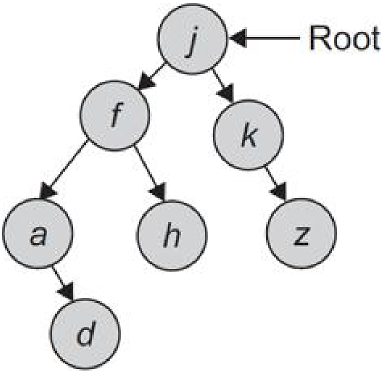
Percurso em profundidade primeiro (depth-first): desce em profundidade na árvore antes de explorar outros nós no mesmo nível.

- Pré-ordem
- In-ordem
- Pós-ordem

Percurso em largura primeiro (breadth-first): percorre os nós nível por nível, geralmente da esquerda para a direita.

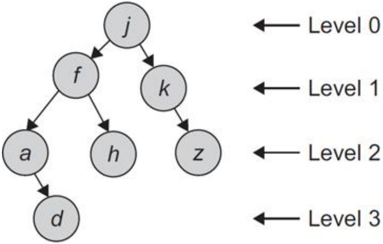
Percurso em Profundidade: Implementado usando Pilha

```
void BinaryTree :: DFS_Tree()
{
   stack S;
   TreeNode *Tmp=Root;
   do
   {
      cout << Tmp->Data;
      if(Tmp->Rchild != Null)
        S.Push(Tmp->Rchild);
      if(Tmp->Lchild != Null)
        S.Push(Tmp->Lchild);
      if(S.IsEmpty()) break;
      Tmp = S.Pop();
   } while(1);
}
```



Percurso em Largura: Implementado usando Fila

```
void BinaryTree :: BFS_Tree()
{
  queue Q;
  TreeNode *Tmp = Root;
  do
  {
    cout << Tmp->Data;
    if ((Tmp->Lchild) != Null)
      Q.Add(Tmp->Lchild);
    if(Tmp->Rchild != Null)
      Q.Add(Tmp->Rchild);
    if(Q.Empty()) break;
    Tmp = Q.Del();
  }
  while(1);
}
```

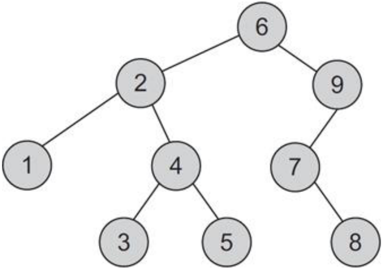


Percurso em Largura e em Profundidade: Exercício

1. Pesquise aplicações para os dois tipos gerais de percurso (em profundidade e em largura). Explique.
2. Pesquise aplicações para os três tipos padrão de percurso em profundidade (pré-ordem, in-ordem, pós-ordem).

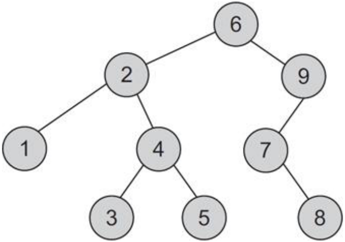
Outras Operações em Árvores
Contagem Total de Nós

```
int   BinaryTree :: CountNode(TreeNode *Root)
{
 if(Root == Null)
  return 0;
 else
  return(1+CountNode(Root->Rchild)+CountNode(Root->Lchild));
}
```



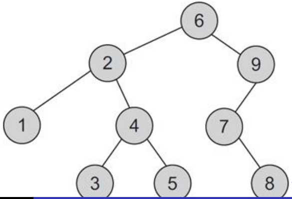
Outras Operações em Árvores
Contagem de Nós Folha

```
int BinaryTree :: CountLeaf(TreeNode *Root)
{
 if(Root == Null)
  return 0;
 else if((Root->Rchild == Null) && (Root->Lchild == Null))
  return(1);
 else
  return(CountLeaf(Root->Lchild) + CountLeaf(Root->Rchild));
}
```



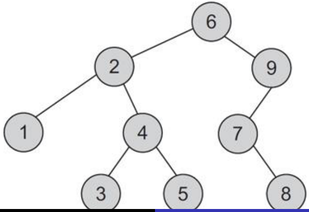
Outras Operações em Árvores
Altura da Árvore

```
int BinaryTree :: TreeHeight(TreeNode *Root)
{
 int heightL, heightR;
 if(Root == Null)
  return 0;
 if(Root->Lchild == Null && Root->Rchild == Null)
  return 0;
 heightL = TreeHeight(Root->Lchild);
 heightR = TreeHeight(Root->Rchild);
 if(heightR > heightL)
  return(heightR + 1);
 return(heightL + 1);
}
```



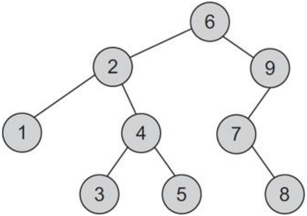
Outras Operações em Árvores
Espelhamento

```
void BinaryTree :: Mirror(TreeNode *Root)
{
 TreeNode *Tmp;
 if(Root != Null)
 {
  Tmp = Root->Lchild;
  Root->Lchild = Root->Rchild;
  Root->Rchild = Tmp;
  Mirror(Root->Lchild);
  Mirror(Root->Rchild);
 }
}
```



Outras Operações em Árvores
Cópia

```
TreeNode *BinaryTree :: TreeCopy()
{
 TreeNode *Tmp;
 if(Root == Null)
  return Null;
 Tmp = new TreeNode;
 Tmp->Lchild = TreeCopy(Root->Lchild);
 Tmp->Rchild = TreeCopy(Root->Rchild);
 Tmp->Data = Root->Data;
 return Tmp;
}
```



Outras Operações em Árvores
Teste de Igualdade

Duas árvores são iguais se:

- Se ambas têm a mesma topologia; e
- Todos os nós correspondentes são iguais (possuem os mesmos valore de dados).

```
int BinaryTree :: BTree_Equal(Binarytree T1, BinaryTree T2)
{
 if(T1.Root == Null && T2.Root == Null)
  return 1;
 if (T1.Root && T2.Root){
  return ((T1.Root->Data == T2.Root->Data) &&
           BTree_Equal(T1.Root->Lchild ,T2.Root->Lchild) &&
           BTree_Equal(T1.Root->Rchild , T2.Root->Rchild));
 }
}
```

Exercícios

1. É possível implementar cada uma as operações descritas acima (contagem de nós e folhas, altura da árvore, espelhamento, cópia e igualdade) em termos das operações de percurso vistas anteriormente? Se não todas, quais?
2. Seria uma forma válida e eficiente de implementar essas operações de maneira não-recursiva?
3. Implemente.

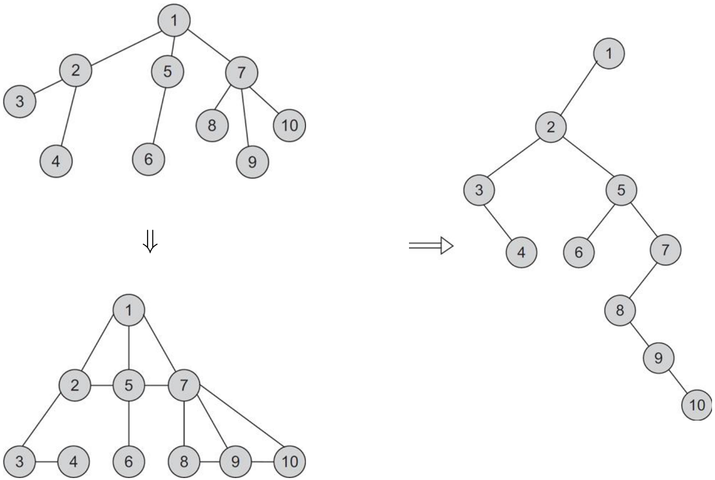
Conversão de Árvores Gerais em Árvores Binárias

Qualquer árvore geral (de grau n) pode ser representada como uma árvore binária.

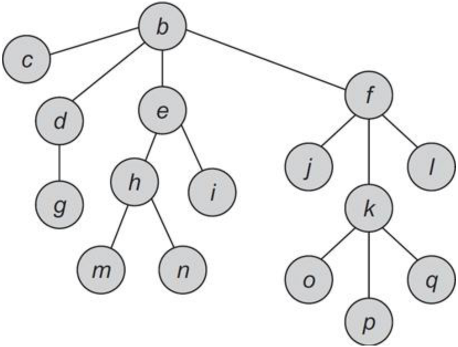
Algoritmo em linhas gerais

1. Todos os nós da árvore geral se tornarão nós da árvore binária.
2. A raiz da árvore geral será a raiz da árvore binária.
3. Sejam os seguintes relacionamentos entre os nós da árvore geral:
    - o primeiro (mais à esquerda) relacionamento filho–pai;
    - o relacionamento entre um nó e seu irmão imediatamente à direita.
4. Conecte cada nó ao seu irmão imediatamente à direita.
    - Obs.: esse irmão irá se tornar o filho da direita do nó.
5. Desconecte cada nó de todos os seus filhos, exceto o filho mais à esquerda.

Conversão de Árvores Gerais em Árvores Binárias
Exemplo 1



Conversão de Árvores Gerais em Árvores Binárias
Exercício



Conversão de Árvores Gerais em Árvores Binárias
Observações

Se a ordem dos filhos em uma árvore não é significativa (ou seja, a árvore é não-ordenada):

- Qualquer dos filhos de um nó pode ser considerado o filho mais à esquerda; e
- Qualquer dos irmãos de um nó pode ser tomado como o irmão imediatamente à direita.

A conversão é reversível

Dada a representação de uma árvore geral na forma de árvore binária, pode-se recriar a árvore geral original.

- O filho da esquerda de um nó torna-se seu filho mais à esquerda
- Um nó que é filho da direita torna-se irmão de seu nó pai

Árvore Binária de Busca: Motivação

Busca Binária

- Realiza a busca em $\mathcal{O}(\log_2 n)$
- Elementos precisam estar ordenados e armazenados posições contíguas (*array*) para permitir acesso direto
- Desvantagem: inserção e remoção têm custo elevado

Uso de listas ligadas

Resolve o problema da inserção e remoção, mas não permite acesso direto – portanto, não serve para busca binária

ABB: Melhor dos dois mundos

Uma forma de implementar uma lista ordenada na qual:

- a busca é rápida (como na busca binária); e
- inserção e remoção também podem ser executadas eficientemente (como em listas ligadas).

Árvore Binária de Busca: Introdução

Propriedades

O valor armazenado em qualquer nó da árvore é:

- maior que qualquer valor armazenado na sub-árvore da esquerda
- menor que qualquer valor armazenado na sub-árvore da direita

$\rightarrow$ As sub-árvores de um nó são também árvores binárias de busca.

Obs.: Assume que os valores (i.e., chaves) armazenados nos nós da árvore são únicos.

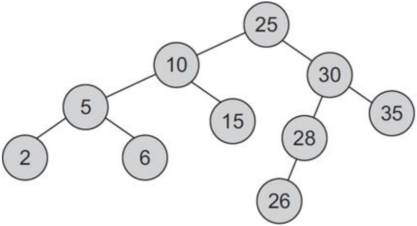
Desempenho

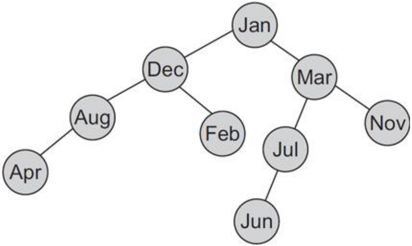
Busca, Inserção, Remoção: $\mathcal{O}(\log_2 n)$
$\rightarrow$ Desde que a árvore seja relativamente balanceada.

Construção

Cria-se uma ABB vazia e insere-se os nós um-a-um.

Árvore Binária de Busca: Exemplos





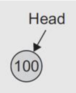
Árvore Binária de Busca: Implementação

```
class TreeNode
{
 <data type> Key;
 TreeNode *Lchild, *Rchild;
};

class BSTree
{
 private:
  TreeNode *Root;
 public:
  BSTree() {Root = NULL;}      // constructor
  void InsertNode(int Key);
  void DeleteNode(int key);
  void Search(int Key);
  bool IsEmpty();
};
```

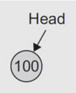
Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

Passo 1: Root = NULL; insere 100

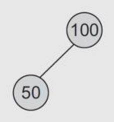


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

Passo 1: Root = NULL; insere 100

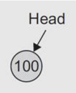


Passo 2: insere 50

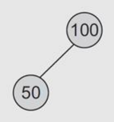


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

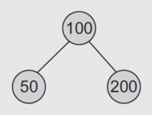
Passo 1: Root = NULL; insere 100



Passo 2: insere 50

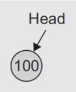


Passo 3: insere 200

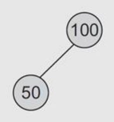


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

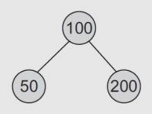
Passo 1: Root = NULL; insere 100



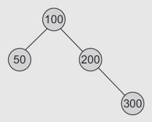
Passo 2: insere 50



Passo 3: insere 200

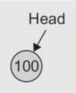


Passo 4: insere 300

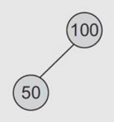


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

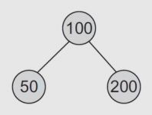
Passo 1: Root = NULL; insere 100



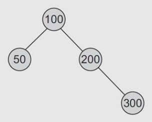
Passo 2: insere 50



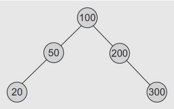
Passo 3: insere 200



Passo 4: insere 300

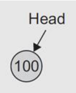


Passo 5: insere 20

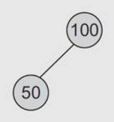


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

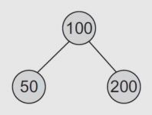
Passo 1: Root = NULL; insere 100



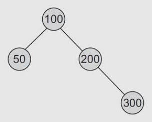
Passo 2: insere 50



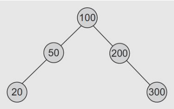
Passo 3: insere 200



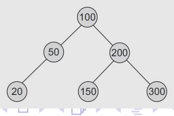
Passo 4: insere 300



Passo 5: insere 20

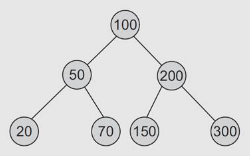


Passo 6: insere 150



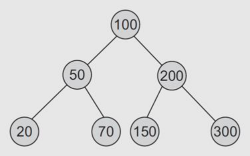
Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

Passo 7: insere 70

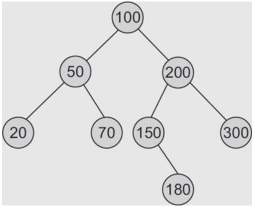


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

Passo 7: insere 70

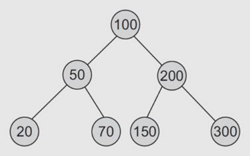


Passo 8: insere 180

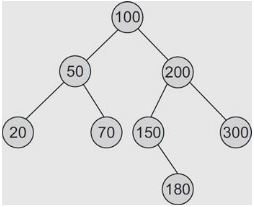


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

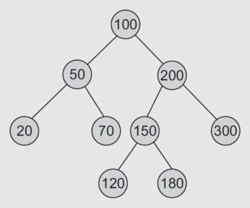
Passo 7: insere 70



Passo 8: insere 180

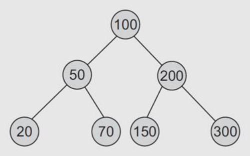


Passo 9: insere 120

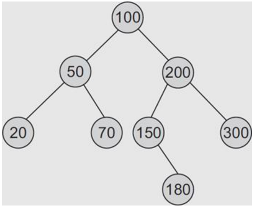


Árvore Binária de Busca: Inserção
Sequência de chaves: 100,50,200,300,20,150,70,180,120,30

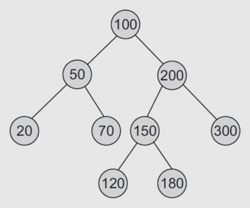
Passo 7: insere 70



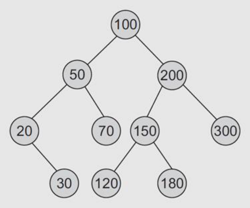
Passo 8: insere 180



Passo 9: insere 120



Passo 10: insere 30



Percursos na árvore binária

Preorder: 100 50 20 30 70 200 150 120 180 300
Inorder: 20 30 50 70 100 120 150 180 200 300
Postorder: 30 20 70 50 120 180 150 300 200 100

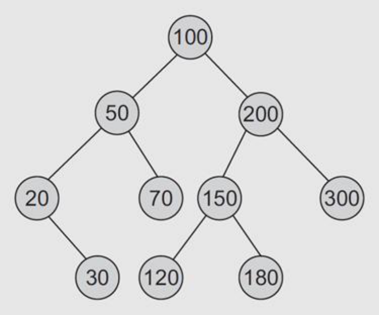


ABB: Inserção

```
TreeNode *BSTree::Insert(int Key){
  TreeNode *Tmp, NewNode = new TreeNode;
  NewNode->Data = Key;
  NewNode->Lchild = NewNode->Rchild = NULL:
  if(Root == NULL){
    Root = NewNode; return;
  }
  Tmp = Root;
  while(Tmp != NUll){
    if(Key < Tmp->Data){
      if(Tmp->Lchild == NULL){
        Tmp->Lchild = NewNode; return;
      }
      Tmp = Tmp->Lchild;
    }
    else{
      if(Tmp->Rchild == Null){
        Tmp->Rchild = NewNode; return;
      }
      Tmp = Tmp->Rchild;
    }
  }
}
```

ABB: Busca

```cpp

TreeNode *BSTree :: Search(int Key)
{
 TreeNode *Tmp = Root;
 while(Tmp){
  if(Tmp->Data == Key)
   return Tmp;
  else if(Key < Tmp->data)
   Tmp = Tmp->Lchild;
  else
   Tmp = Tmp->Rchild;
 }
 return NULL;
}
```

Exercício: Implementar a busca recursiva.

ABB: Remoção

Três casos:

- Caso 1: Nó folha
- Caso 2: Nó com uma sub-árvore
- Caso 3: Nó com duas sub-árvores

ABB: Remoção – Caso 1: Nó folha

Busca pelo nó e o remove: atribui NULL ao ponteiro do nó pai (esquerda ou direita); libera a memória ocupada pelo nó.

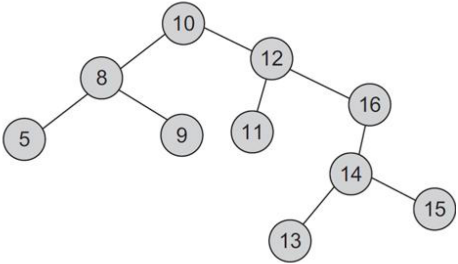
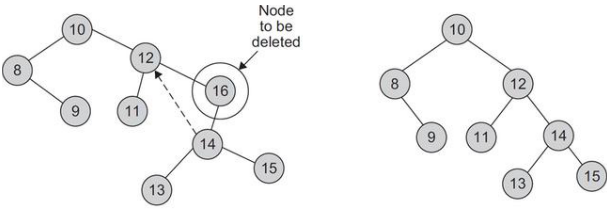


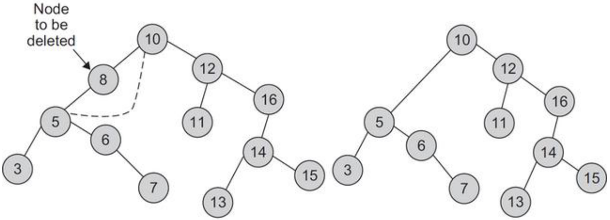
ABB: Remoção – Caso 2: Nó com uma única sub-árvore

Caso 2a: Sub-árvore da esquerda
Caso 2b: Sub-árvore da direita

ABB: Remoção – Caso 2a
Nó com apenas a sub-árvore da esquerda



(a)



(b)

ABB: Remoção – Caso 2b
Nó com apenas a sub-árvore da direita

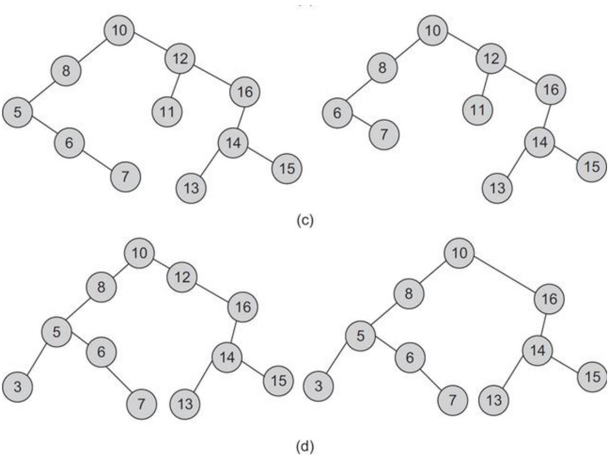
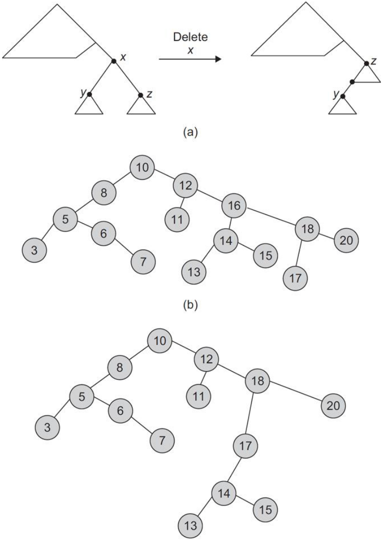


ABB: Remoção – Caso 3 (abordagem 1)

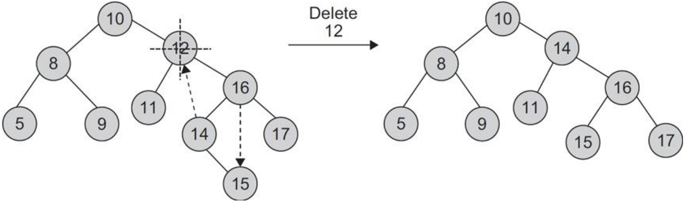


(a) (b) (c)

ABB: Remoção – Caso 3 (abordagem 2)

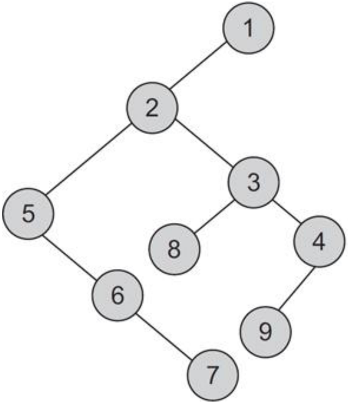
Substitui o nó a ser removido pelo seu sucessor (obtido pelo percurso in-ordem na sub-árvore da direita)

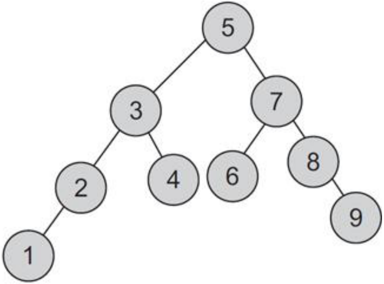
O filho da direita do sucessor (se houver) torna-se filho da esquerda de seu avô



P: Alguma diferença em se tratando da remoção de um nó que é filho da esquerda?

Árvore Binária vs. Árvore Binária de Busca





Diferenças em relação às operações fundamentais: Inserção, Busca, Remoção, Percurso

Árvores Binárias Alinhavadas (Threaded Binary Trees)

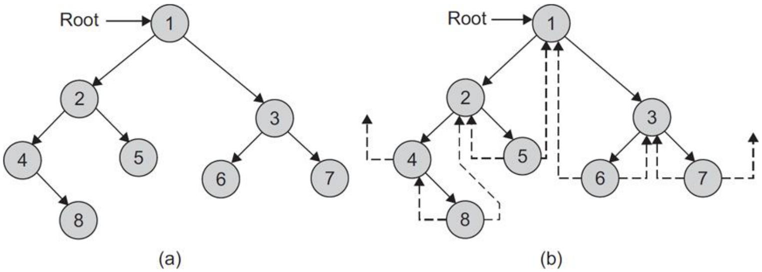
Motivação:

- Dar uma utilidade para os ponteiros nulos dos nós da árvore.
- Tornar mais eficientes o percurso na árvore: $\mathcal{O}(\log_2 n)$ para percurso ascendente, sem uso de estrutura de dados auxiliar (pilha ou fila) ou recursão.

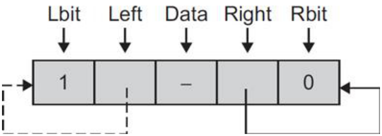
Ideia geral:

- Faz o ponteiro nulo da esquerda apontar para o predecessor do nó.
- Faz o ponteiro nulo da direita apontar para o sucessor do nó.
- Diferencia o significado de cada ponteiro usando um bit auxiliar:
  - 0: link para nó filho
  - 1: thread (aponta para sucessor ou predecessor, dependendo do caso)
- links têm finalidade estrutural; threads têm finalidade acessória (tornar o percurso mais eficiente)

Threaded Binary Trees – Exemplo

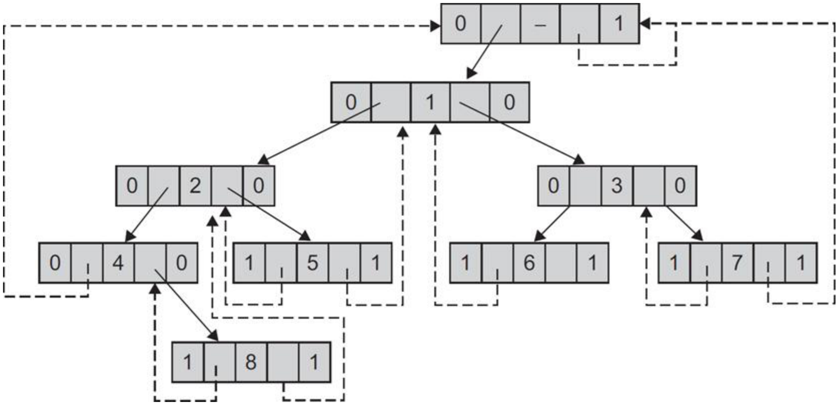


Threaded Binary Trees – Estrutura dos Nós




```
Class TBTNode
{
   boolean Lbit, Rbit;
  <Datatype> Data;
  TBTNode *Left, *Right;
};
```

Threaded Binary Trees – Representação



Threaded Binary Trees – Construção (Método Top Down)

Começar com uma árvore binária dada.

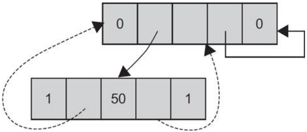
1. Percorrer a árvore em largura e, para cada nó não-folha visitado, ajustar os ponteiros (threads) que apontam para este nó a partir de outros dois nós da árvore:
   - predecessor – nó mais à direita da sub-árvore da esquerda.
   - sucessor – nó mais à esquerda da sub-árvore da direita.
2. Ajustar o thread do nó mais à esquerda da árvore para apontar para o nó fictício (head)
3. Ajustar o thread do nó mais à direita da árvore para apontar para o nó fictício (head).
4. Obs.: Nós folha não precisam ser processados.

P: Por que não uma abordagem bottom-up, onde se identifica primeiro os ponteiros NULL e os faz apontar para os nós apropriados?

Threaded Binary Trees – Construção (Método por Inserção)

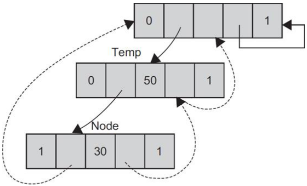
Costurar (thread) a árvore à medida em que os nós são inseridos nela.

1: Insere a raiz (valor 50) e faz seus dois threads apontarem para o nó head. LBit = RBit = 1



2: Insere um novo nó com valor de chave 30:

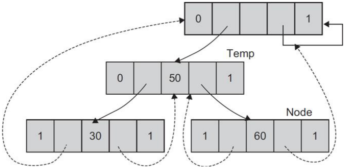
```
Node->left = Temp->left
Temp->left = Node
Temp->LBit = 0
Node->right = Temp
Node->LBit = Node->RBit = 1
```



Threaded Binary Trees – Construção (Método por Inserção)

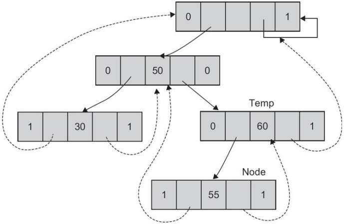
3: Insere um novo nó com valor de chave 60:

```
Node->right = Temp->right
Temp->right = Node
Temp->RBit = 0
Node->left = Temp
Node->RBit = Node->LBit = 1
```



4: Insere um novo nó com valor de chave 55:

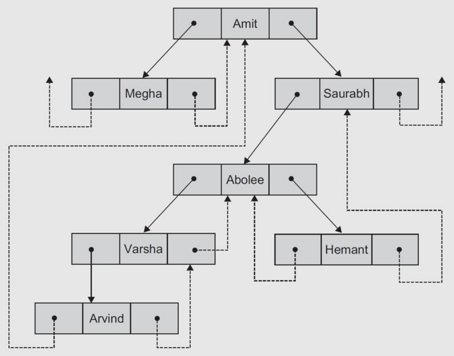
```
Node->left = Temp->left
Temp->left = Node
Temp->LBit = 0
Node->right = Temp
Node->RBit = Node->LBit = 1
```



Threaded Binary Trees – Percurso

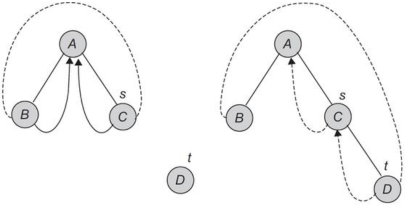
Não requer o uso de estruturas auxiliares. Apenas segue-se os ponteiros (links e threads).

Exemplo – Árvore cujo percurso in-ordem resulta na sequência de chaves: `Megha, Amit, Arvind, Varsha, Abolee, Hemant, Saurabh.`



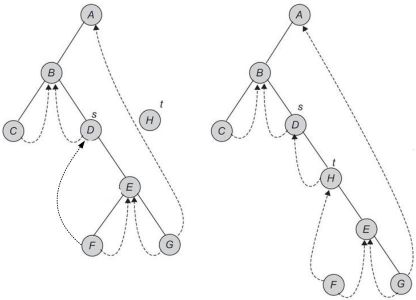
Threaded Binary Trees – Inserção

Exemplo 1: Inserção de nó folha à direita.



Threaded Binary Trees – Inserção

Exemplo 2: Inserção de nó interno (não-folha) à direita.



Threaded Binary Trees – Remoção

Exemplo: Remoção do nó com valor de chave 'D'