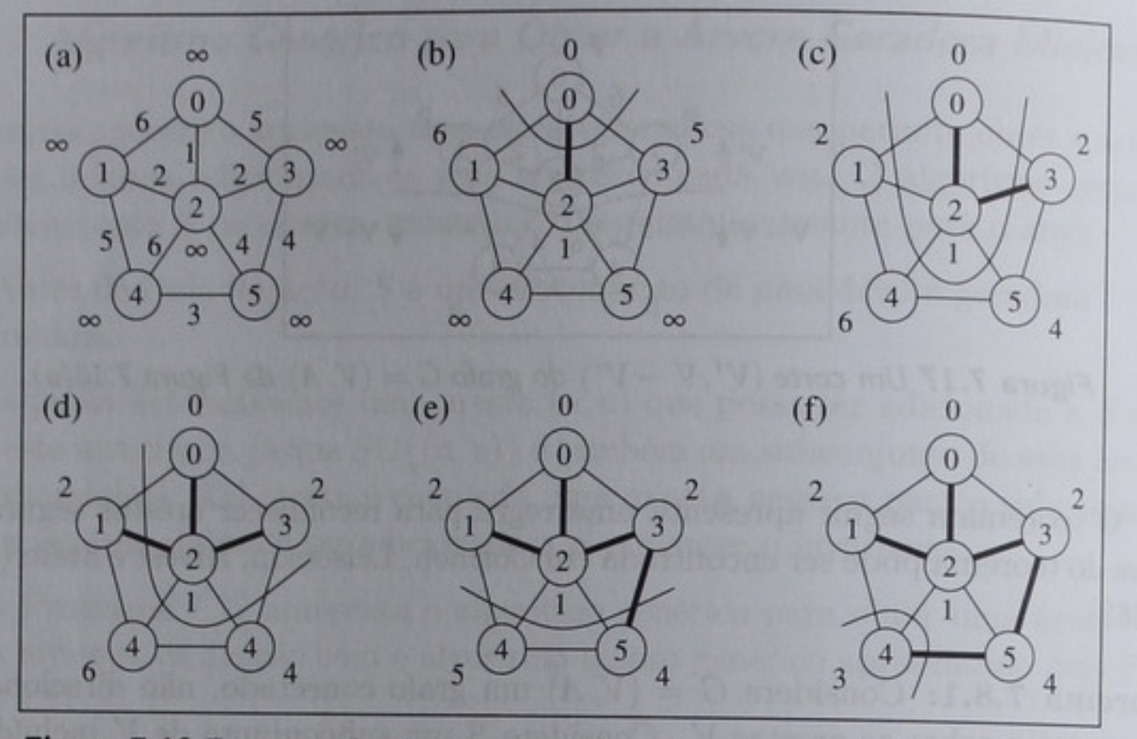

Figura 7.18 *Execução do algoritmo de Prim sobre o grafo da Figura 7.16(a).*

Para obter uma boa implementação para o algoritmo de Prim, é preciso realizar de forma eficiente a seleção de uma nova aresta a ser adicionada à árvore formada pelas arestas em S. Durante a execução do algoritmo, todos os vértices que não estão na árvore geradora mínima residem em uma fila de prioridades A baseada no campo p e implementada como um *heap* (vide Seção 4.1.5). Assim, para cada vértice v, $p[v]$ é a aresta de menor peso conectando v a um vértice na árvore. Como o *heap* utilizado mantém na árvore os vértices, mas a condição do *heap* é mantida pelo peso da aresta por meio do arranjo $p[v]$, o *heap* é indireto, conforme pode ser observado no procedimento RefazInd do Programa 7.18. O arranjo $Pos[v]$ fornece a posição do vértice v dentro do *heap* A, permitindo assim que o vértice v possa ser acessado a um custo $O(1)$. O acesso ao vértice v é necessário para a operação DiminuiChaveInd, operação similar à operação AumentaChave realizada pelo Programa 4.12 na Seção 4.1.5.

O Programa 7.19 implementa o algoritmo de Prim. O procedimento AgmPrim recebe como entrada o grafo G e o vértice $Raiz$. O campo Antecessor[v] armazena o antecessor de v na árvore. Durante a execução do algoritmo, o subconjunto S do algoritmo GenericoAgm do Programa 7.17 é mantido de forma implícita como

$$S = \{(v, Antecessor[v]) : v \in V - \{Raiz\} - A\}.$$

Quando o algoritmo termina, a fila de prioridades A está vazia, e a árvore geradora mínima S para G é

$$S = \{(v, Antecessor[v]) : v \in V - \{Raiz\}\}.$$

Programa 7.18 *Operadores para manter o heap indireto*

```
{-- Entra aqui o operador Constroi da Seção 4.1.5 (Programa 4.10)   --}
{-- Trocando a chamada Refaz (Esq, n , A) por RefazInd (Esq, n, A) --}
procedure RefazInd (Esq, Dir: TipoIndice; var A: TipoVetor);
label 999;
var i: TipoIndice; j: integer; x: TipoItem;
begin
  i := Esq; j := 2 * i; x := A[i];
  while j <= Dir do
    begin
    if j < Dir
    then if p[A[j].Chave] > p[A[j + 1].Chave] then j := j + 1;
    if p[x.Chave] <= p[A[j].Chave] then goto 999;
    A[i] := A[j]; Pos[A[j].Chave] := i; i := j; j := 2 * i;
    end;
  999: A[i] := x; Pos[x.Chave] := i;
end; { RefazInd }

function RetiraMinInd (var A: TipoVetor): TipoItem;
begin
  if n < 1
  then writeln ('Erro: heap vazio')
  else begin
       RetiraMinInd := A[1]; A[1] := A[n];
       Pos[A[n].chave] := 1;
       n := n - 1;
       RefazInd (1, n, A);
       end;
end; { RetiraMinInd }

procedure DiminuiChaveInd (i: TipoIndice; ChaveNova: TipoPeso;
                           var A: TipoVetor);
var x: TipoItem;
begin
  if ChaveNova > p[A[i].Chave]
  then writeln ('Erro: ChaveNova maior que a chave atual')
  else begin
       p[A[i].Chave] := ChaveNova;
       while (i>1) and (p[A[i div 2].Chave] > p[A[i].Chave]) do
         begin
         x := A[i div 2];
         A[i div 2] := A[i];
         Pos[A[i].Chave] := i div 2;
         A[i] := x;
         Pos[x.Chave] := i;
         i := i div 2;
         end;
       end;
end; { DiminuiChaveInd }
```