Universidade de São Paulo
Escola de Artes, Ciências e Humanidades
Sistemas de Informação

ACH2023 - ALGORITMOS E ESTRUTURAS DE DADOS I

Willian Yukio Honda
Ivandré Paraboni

atualizada em 18/11/2016

Esta apostila encontra-se eternamente em fase de elaboração.
Por favor comunique eventuais erros, críticas ou sugestões escrevendo para ivandre@usp.br

1. Introdução

Esta apostila apresenta uma coletânea de algoritmos sobre uma variedade de estruturas de dados de memória principal estudadas na disciplina ACH2023 – Algoritmos e Estruturas de Dados I, a saber:

Listas Lineares
- Sequenciais
- Ligadas
  - Implementação Estática
  - Implementação Dinâmica
  - Técnicas Especiais: Cabeça, Sentinela, Circularidade, Encadeamento Duplo
- Filas
  - Implementação Estática
  - Implementação Dinâmica
- Deques (Filas de duas Pontas)
  - Implementação Dinâmica
- Pilhas
  - Implementação Estática
  - Implementação Dinâmica
  - Implementação de Múltiplas Pilhas em um Vetor
  - Aplicações
- Matrizes Esparsas
- Listas Generalizadas

Listas não Lineares
- Árvores Binárias
- Árvores de Busca Binária
- Árvores AVL

1.1. Notação utilizada

Para simplificar o código apresentado ao longo desta apostila, tornando-o o mais genérico possível, pressupõe-se a definição a seguir:

```
// tamanho máximo do vetor estático
# define MAX 50
```

2. Listas Lineares

Uma lista linear é uma série de elementos ordenados na qual cada elemento exceto o primeiro possui um e apenas um antecessor, e cada elemento exceto o último possui um e apenas um sucessor.

2.1. Listas Lineares Sequenciais

É a lista linear na qual a ordem (lógica) dos elementos da lista coincide com sua posição física (em memória). Ou seja, elementos adjacentes da lista ocupam posições contíguas de memória.

A forma mais comum de implementação de uma lista sequencial é através de um vetor de elementos (A) do tipo REGISTRO de tamanho máximo possível definido pela constante MAX.

O vetor é conjugado a um contador de número de posições efetivamente ocupadas (nroElem). A ocupação do vetor se dá sempre em posições contíguas, ou seja, nroElem-1 indica o último elemento existente na estrutura.

```
typedef struct  {
        REGISTRO A[MAX];
        int nroElem;
} LISTA;
```

Os registros contêm campos de informações variadas que são dependentes da aplicação. Por exemplo, um registro de aluno conteria campos como nome, idade, NroUSP etc. Para efeito de demonstração da busca em estruturas ordenadas e outras operações de identificação de elementos, definimos também um campo chave em cada registro.

```
typedef struct  {
        int chave;
        // outros campos...
} REGISTRO;
```

Vantagens:

– Acesso direto a qualquer elemento com base no seu índice. O tempo é constante O(1). No entanto, em muitas aplicações o índice do dado procurado não é conhecido, o que faz desta uma vantagem apenas relativa.

– Se a lista estiver ordenada pela chave em questão, a busca por uma chave pode ser efetuada através de busca binária O(lg n), que é excelente.

Desvantagens:

– Mesmo em uma estrutura ordenada, o pior caso de inserção e exclusão (na frente da lista) exige movimentação de todos os n elementos da lista, ou seja, O(n). Isso pode ser inadequado para aplicações em que ocorrem muitas atualizações, e é principalmente por causa disso que a lista sequencial é muitas vezes substituída por uma estrutura de dados mais complexa.

– Implementação estática, exigindo que o tamanho do vetor seja previamente estabelecido. Embora algumas linguagens de programação (como C) permitam a expansão posterior de um vetor deste tipo, esta operação requer cópia de áreas de dados inteiras em memória, a um custo O(n) que pode inviabilizar sua aplicação no caso geral.

```
// Inicialização da lista sequencial
void inicializarListaSequencial(LISTA *l) {
      l->nroElem = 0;
}

// Exibição da lista sequencial
void exibirLista(LISTA l) {
      int i;
      for (i=0; i < l.nroElem; i++)
          printf("%d ", l.A[i].chave);
}

// Retornar o tamanho da lista sequencial
int tamanho(LISTA l) {
      return(l.nroElem);
}

// Retornar a chave do primeiro elemento da lista sequencial
int primeiroElem(LISTA l) {
      if(l.nroElem > 0) return(l.A[0].chave);
      else return(-1); // lista vazia
}

// Retornar a chave do ultimo elemento da lista sequencial
int ultimoElem(LISTA l) {
      if(l.nroElem > 0) return(l.A[l.nroElem-1].chave);
      else return(-1); // lista vazia
}

// Retornar a chave do n-ésimo elemento da lista sequencial
int enesimoElem(LISTA l, int n) {
      if(n <= l.nroElem) return(l.A[n-1].chave);
      else return(-1);
}

// Destruição da lista sequencial
void destruirLista(LISTA *l) {
      l->nroElem = 0;
}
```

```
// Inserção "direta" na posição i
bool inserirElemListaSeq(int ch, int i, LISTA *l) {
      int j;
      if((l->nroElem >= MAX) || (i < 0) || (i > (l->nroElem)))
            return(false); // lista cheia ou índice inválido
      if((l->nroElem > 0) && (i < l->nroElem))
            for (j = l->nroElem; j >= (i+1); j--)
                  l->A[j] = l->A[j-1];
      l->A[i].chave=ch;
      l->nroElem++;
      return(true);
}

// Busca sequencial em lista (ordenada por chave ou não)
int buscaSeq(int ch, LISTA l) {
      int i = 0;
      while (i < l.nroElem)   {
          if(ch == l.A[i].chave) return(i); // achou
          else i++;
      }
      return(-1);  // não achou
}

// Busca sequencial em lista COM SENTINELA (vetor criado com MAX+1 posições)
int buscaSent(int ch, LISTA l) {
      int i = 0;
      l.A[l.nroElem].chave = ch; // sentinela
      while(l.A[i].chave < ch) i++;
      if((i > (l.nroElem - 1)) || (l.A[i].chave != ch)) return(-1); // não achou
      else return(i);
}

// Busca binária em lista ordenada
int buscaBin(int ch, LISTA l) {
      int inf, sup, meio;
      inf = 0;
      sup = l.nroElem - 1;
      while(inf <= sup) {
                  meio = ((inf + sup) / 2);
                  if(l.A[meio].chave == ch) return(meio);  // achou
                  else  {
                        if(l.A[meio].chave < ch) inf = meio + 1;
                        else sup = meio - 1;
                   }
      }
      return(-1);
}
```

```
// Inserção em lista ordenada COM SENTINELA sem duplicação
bool inserirElemListaOrd(int ch, LISTA *l) {
      int i = 0;
      if(l->nroElem >= MAX) return(false); // lista cheia
      l->A[l->nroElem].chave = ch; // sentinela
      while(l->A[i].chave < ch) i++;
      if((l->A[i].chave == ch) && (i < l->nroElem))
            return(false);
      else return(inserirElemListaSeq(ch, i, l));
}

// Exclusão
bool excluirElemLista(int ch, LISTA *l) {
      int pos, j;
      pos = buscaSeq(ch, *l);
      if(pos == -1) return(false); // não existe
      for(j = pos; j < l->nroElem - 1; j++)
            l->A[j] = l->A[j+1];
      l->nroElem--;
      return(true);
}
```

2.2. Listas Lineares Ligadas (ou Encadeadas)

Se a inserção e exclusão em listas sequenciais podem acarretar grande movimentação de dados, uma solução óbvia é permitir que os dados ocupem qualquer posição disponível (e não necessariamente a posição física "correta"), e então criar um esquema para preservar a ordem dos elementos e gerenciamento de nós livres / ocupados.

Uma lista linear ligada (ou simplesmente lista ligada) é uma lista linear na qual a ordem (lógica) dos elementos da lista (chamados "nós") não necessariamente coincide com sua posição física (em memória). Pode ser implementada de forma estática (usando-se um vetor) ou, em linguagens de programação que oferecem suporte à alocação dinâmica, com uso de ponteiros.

2.2.1. Listas Ligadas de Implementação Estática

A lista é formada pelo vetor de registros (A), um indicador de início da estrutura (inicio) e um indicador de início da lista de nós disponíveis (dispo). Na prática, inicio e dispo são as entradas de duas listas que compartilham o mesmo vetor, sendo uma para os elementos efetivos da lista, e a outra para armazenar as posições livres.

```
typedef struct  {
      REGISTRO A[MAX];
      int inicio;
      int dispo;
} LISTA;
```

Cada registro contém, além dos campos exigidos pela aplicação, um campo prox que contém um índice para o próximo elemento na série (vazio ou ocupado, conforme descrito a seguir). Um campo prox com valor -1 será usado para designar que o elemento em questão não possui sucessor.

```
typedef struct  {
      int chave;
      int prox;
} REGISTRO;
```

Ao ser criada, a lista possui inicio = -1 (que indica que a lista está vazia) e dispo = 0 (ou seja, a primeira posição do vetor está disponível). Além disso, os campos prox de cada registro (exceto o último) apontam para o registro seguinte, constituindo uma lista de registros vazios encabeçada por dispo. O campo prox do último registro recebe o valor -1 indicando que não há mais elementos depois daquele ponto.

A lista está cheia quando não há mais nós disponíveis (i.e., quando dispo == -1). A lista esta vazia quando não há elemento inicial (i.e., quando inicio == -1).

```
// Inicialização
void inicializarListaLigadaEstatica(LISTA *l) {
      l->inicio = -1;
      l->dispo = 0;
      for(int i=0; i < MAX - 1; i++)
            l->A[i].prox = i + 1;
      l->A[MAX - 1].prox = -1;
}
```

```
// Exibição da lista completa
void exibirLista(LISTA l) {
    int i = l.inicio;
    while (i > -1)  {
          printf("%d ", l.A[i].chave);
          i = l.A[i].prox;
    }
}

// Busca sequencial, retornando a posição da chave e do anterior
int buscaSeqOrd(int ch, LISTA l, int *ant) {
      int i = l.inicio;
      *ant= -1;
      while (i != -1)   {
            if(l.A[i].chave >= ch) break;
            *ant = i;
            i= l.A[i].prox;
      }
      if(i==-1) return -1;
      If(l.A[i].chave==ch) return(i);
      else return -1;
}
```

O gerenciamento de nós livres e ocupados exige a criação de rotinas específicas para “alocar” e “desalocar” um nó da lista apontada por dispo. A alocação envolve descobrir o índice de uma posição válida no vetor na qual novos dados possam ser inseridos, além de retirar esta posição da lista de disponíveis. A desalocação envolve a devolução de uma posição à lista de disponíveis para que possa ser reutilizada.

As rotinas de alocação/desalocação não devem ser chamadas sem que sejam seguidas da correspondente inserção/exclusão, pois haverá perda de dados e a estrutura se tornará inconsistente.

```
// Obter nó disponível - a lista é alterada
int obterNó(LISTA *l) {
    int result = l->dispo;
    if(l->dispo > -1) {
            l->dispo = l->A[l->dispo].prox;
    }
    return(result);
}

// Devolver nó p/ dispo – a lista é alterada
void devolverNo(LISTA *l, int j) {
      l->A[j].prox= l->dispo;
      l->dispo = j;
}
```

```
// Exclusão do elemento de chave indicada
bool excluirElemListaEnc(int ch, LISTA *l) {
    int ant, i;
    i = buscaSeqOrd(ch, *l, &ant);
    if(i < 0) return(false);
    if(ant == -1) l->inicio = l->A[i].prox;
    else l->A[ant].prox = l->A[i].prox;
    devolverNo(l, i);
    return(true);
}

// inserção em lista ordenada sem duplicações
bool inserirElemListaEncOrd(int ch, LISTA *l) {
      int ant, i;
      i = buscaSeqOrd(ch, *l, &ant);
      if((l->dispo < 0) || (i != -1)) return(false);
      i = ObterNo(l);
      l->A[i].chave = ch;
      if(l->inicio<0)   {
            // inserção do primeiro elemento de lista vazia
            l->inicio = i;
            l->A[i].prox = -1;
      }
      else  {
            if(ant < 0) {
                  // inserção no início de lista já existente
                  l->A[i].prox = l->inicio;
                  l->inicio = i;
            }
            else  {
                  // inserção entre dois elementos
                  l->A[i].prox = l->A[ant].prox;
                  l->A[ant].prox = i;
            }
      }
      return(true);
}
```

2.2.2. Listas Ligadas de Implementação Dinâmica

Para evitar a necessidade de definição antecipada do tamanho máximo da estrutura de implementação estática (i.e., o vetor), podemos tirar proveito dos recursos de alocação dinâmica de memória das linguagens de programação como C, deixando o gerenciamento de nós livres / ocupados a cargo do ambiente de programação. Esta técnica constitui à implementação dinâmica de listas ligadas, e requer o uso das funções disponibilizadas em malloc.h:

```
# include <malloc.h>
```

Em uma lista ligada de implementação dinâmica, não há mais uso de vetores. Cada elemento da lista é uma estrutura do tipo NO, que contém os dados de cada elemento (inclusive a **chave**) e um ponteiro **prox** para o próximo nó da lista. Um nome auxiliar (estrutura) é usado para permitir a auto-referência ao tipo NO que está sendo definido.

```
typedef struct estrutura {
        int chave;
        int info;
        estrutura *prox;
} NO;
```

O tipo LISTA propriamente dito é simplesmente um ponteiro inicio apontando para o primeiro nó da estrutura (ou para NULL no caso da lista vazia). O último elemento da lista possui seu ponteiro prox também apontando para NULL. Embora não seja necessário o encapsulamento deste ponteiro em um tipo LISTA (afinal, uma lista é apenas um ponteiro, que pode ser NULL ou não), esta medida será adotada aqui por questões de padronização em relação aos tipos LISTA anteriores, e também porque alguns dos próximos tipos agregarão mais informações a esta definição.

```
typedef struct {
      NO* inicio;
} LISTA;
```

A alocação e desalocação de nós é feita dinamicamente pelo próprio compilador C através das primitivas malloc e free, respectivamente.

```
NO* p = (NO*) malloc(sizeof(NO));   // cria um novo nó em memória, apontado por p
free(p);                            // a área de memória apontada por p é liberada;
```

Via de regra, *malloc*() é usado em rotinas de inserção ou criação de novos nós, enquanto que *free*() é usado na exclusão. Rotinas que não criam ou destroem nós dificilmente precisam usar estas funções.

A única diferença significativa entre as implementações estática e dinâmica de listas ligadas está no fato de que a implementação estática "simula" uma lista ligada em vetor, e nos obriga a gerenciar as posições livres e ocupadas. Isso deixa de ser necessário no caso da alocação dinâmica "real".

```
// Inicialização
void inicializarLista(LISTA *l) {
     l->inicio = NULL;
}

// Exibição da lista completa
void exibirLista(LISTA l) {
      NO* p = l.inicio;
      while (p)   {
          printf("%d ",p->chave);
          p = p->prox;
      }
}

// Retornar o primeiro elemento da lista
NO* primeiroElemLista(LISTA l) {
     return(l.inicio);
}

// Retornar o último elemento da lista
NO* ultimoElemLista(LISTA l) {
      NO* p = l.inicio;
      if(p)
         while(p->prox) p = p->prox;
      return(p);
}

// Retornar o enésimo elemento da lista
NO* enesimoElemLista(LISTA l, int n) {
   NO* p = l.inicio;
   int i = 1;
   if(p)
      while((p->prox)&&(i<n))
      {
         p = p->prox;
         i++;
      }
   if(i != n) return(NULL);
   else return(p);
}
```

```
// Quantos elementos existem na lista
int tamanhoLista(LISTA l) {
      NO* p = l.inicio;
      int tam = 0;
      while (p)   {
            tam++;
            p = p->prox;
      }
      return(tam);
}

// Busca pela chave ch na lista (ordem crescente) retornando p e ant
NO* buscaSeqOrd(int ch, LISTA l, NO* *ant) {
      NO* p = l.inicio;
      *ant = NULL;
      while(p)    {
         if(p->chave >= ch) break;
         *ant = p;
         p = p->prox;
      }
      if(p)
            if(p->chave == ch)
                  return(p);
      return(NULL);
}

// Inserção da chave ch na lista ordenada sem duplicações
bool inserirElemListaOrd(int ch , LISTA *l)    {
      NO* novo;
      NO* ant;
      novo = buscaSeqOrd(ch, *l, &ant);
      if(novo) return(false);
      novo = (NO*) malloc(sizeof(NO));
      novo->chave = ch;
      if(!l->inicio)    {     // 1a. inserção em lista vazia
            l->inicio = novo;
            novo->prox = NULL;
      }
      else  {
            if(!ant) {        // inserção no início da lista
                  novo->prox  = l->inicio;
                  l->inicio = novo;
            }
            else {            // inserção após um nó existente
                  novo->prox = ant->prox;
                  ant->prox = novo;
            }
      }
      return(true);
}
```

```
// Anexar novo elemento ao final da lista, duplicado ou não
void anexarElemLista(int ch, LISTA *l) {
     NO* novo;
     NO* ant;
     ant = ultimoElemLista(*l);
     novo = (NO *) malloc(sizeof(NO));
     novo->chave = ch;
     novo->prox = NULL;
     if(!ant) l->inicio = novo;
     else ant->prox = novo;
}

// Exclusão da chave dada
bool excluirElemLista(int ch, LISTA *l) {
    NO* ant;
    NO* elem;
    elem = buscaSeqOrd(ch, *l, &ant);
    if(!elem) return(false);              // nada a excluir
    if(!ant) l->inicio = elem->prox;      // exclui 1o. elemento da lista
    else ant->prox = elem->prox;          // exclui elemento que possui ant
    free(elem);                           // exclusão "física"
    return(true);
}

// Destruição da lista
void destruirLista (LISTA *l) {
    NO* atual;
    NO* prox;
    atual = l->inicio;
    while (atual) {
          prox = atual->prox;       // guarda próxima posição
          free(atual);              // libera memória apontada por atual
          atual = prox;
     }
     l->inicio = NULL;              // ajusta início da lista (vazia)
}
```

Listas dinâmicas com nó sentinela

O nó sentinela, criado ao final de uma lista, é usado para armazenar a chave de busca e assim acelerar o respectivo algoritmo, uma vez que reduz o número de comparações necessárias pela metade.

```
typedef struct {
      NO* inicio;
      NO* sentinela;
} LISTA;

// Inicialização (lista com sentinela)
void inicializarLista(LISTA *l) {
     l->sentinela = (NO*) malloc(sizeof(NO));
     l->inicio = l->sentinela;
}
```

```
// Exibição (lista com sentinela)
void exibirLista(LISTA l) {
      NO* p = l.inicio;
      while (p != l.sentinela) {
            printf("%d ",p->chave); // chave deve ser int
            p = p->prox;
      }
}

// Primeiro elemento da lista com sentinela
NO* primeiroElemLista(LISTA l) {
      if(l.inicio == l.sentinela)
            return(NULL);
      else
            return(l.inicio);
}

// Último elemento da lista com sentinela
NO* ultimoElemLista(LISTA l) {
   NO* p = l.inicio;
   if(p == l.sentinela) return(NULL);
   while(p->prox != l.sentinela)
            p = p->prox;
   return(p);
}

// N-ésimo elemento da lista com sentinela
NO* enesimoElemLista(LISTA l, int n) {
      NO* p = l.inicio;
      int i = 1;
      if(p == l.sentinela) return(NULL);
      while((p->prox != l.sentinela) && (i < n)) {
            p = p->prox;
            i++;
      }
      if(i != n) return(NULL);
      else return(p);
}
```

```
// Quantos elementos existem na lista com sentinela
int tamanhoLista(LISTA l) {
      NO* p = l.inicio;
      int tam = 0;
      while (p != l.sentinela) {
            tam++;
            p = p->prox;
      }
      return(tam);
}

// Busca da chave em lista ordenada e com sentinela
NO* buscaSeqOrd(int ch, LISTA l, NO* *ant) {
      NO* p = l.inicio;
      *ant = NULL;
      l.sentinela->chave = ch;
      while(p->chave < ch)    {
            *ant = p;
            p = p->prox;
      }
      if((p != l.sentinela) && (p->chave == ch)) return(p);
      else return(NULL);
}

// Inserção em lista ordenada sem repetição com sentinela
bool inserirElemListaOrd(int ch , LISTA *l) {
      NO* novo;
      NO* ant;
      novo = buscaSeqOrd(ch, *l, &ant);
      if(novo) return(false);
      novo = (NO*) malloc(sizeof(NO));
      novo->chave = ch;
      if(l->inicio == l->sentinela) {
            l->inicio = novo;
            novo->prox = l->sentinela;
      }
      else  {
            if(ant==NULL) {
                  novo->prox  = l->inicio;
                  l->inicio = novo;
            }
            else  {
                  novo->prox = ant->prox;
                  ant->prox = novo;
            }
    }
    return(true);
}
```

```
// Anexar um novo elemento à lista com sentinela
void anexarElemLista(int ch, LISTA *l) {
     NO* novo;
     NO* ant;
     ant = ultimoElemLista(*l);
     novo = (NO *) malloc(sizeof(NO));
     novo->chave = ch;
     novo->prox = l->sentinela;
     if(ant == NULL) l->inicio = novo;
     else ant->prox = novo;
}

// Destruição da lista com sentinela
void destruirLista (LISTA *l) {
      NO* atual;
      NO* prox;
      atual = l->inicio;
      while (atual != l->sentinela ) {
                prox = atual->prox;
                free(atual);
                atual = prox;
      }
      l->inicio = l->sentinela;
}
```

Listas dinâmicas com nó cabeça e circularidade

O nó cabeça, criado no início de uma lista, é usado para simplificar o projeto dos algoritmos de inserção e exclusão. Pode também armazenar a chave de busca (funcionando como nó sentinela) quando a lista for circular.

A circularidade é em geral exigência da aplicação (e.g., que precisa percorrer continuamente a estrutura) mas pode também facilitar inserções e exclusões quando combinada com uso de um nó cabeça.

Os algoritmos a seguir são baseados em listas circulares com nó cabeça.

```
// Inicialização da lista circular e com nó cabeça
void inicializarLista(LISTA *l) {
     l->cabeca = (NO*) malloc(sizeof(NO));
     l->cabeca->prox = l->cabeca;
}
```

```
// Exibição da lista circular e com nó cabeça
void exibirLista(LISTA l) {
      NO* p = l.cabeca->prox;
      while (p!=l.cabeca) {
            printf("%d ",p->chave); // deve ser int
            p = p->prox;
    }
}

// 1º. elemento da lista
NO* primeiroElemLista(LISTA l) {
     if(l.cabeca->prox == l.cabeca) return(NULL);
     else return(l.cabeca->prox);
}

// Último elemento da lista
NO* ultimoElemLista(LISTA l) {
   NO* p = l.cabeca->prox;
   if(p == l.cabeca) return(NULL);
   while(p->prox!=l.cabeca)
            p = p->prox;
   return(p);
}

// N-ésimo elemento da lista
NO* enesimoElemLista(LISTA l, int n) {
      NO* p = l.cabeca->prox;
      int i = 1;
      while( (p->prox != l.cabeca) && (i < n) ) {
            p = p->prox;
            i++;
      }
      if(i != n) return(NULL);
      else return(p);
}


// Quantos elementos existem?
int tamanhoLista(LISTA l) {
      NO* p = l.cabeca->prox;
      int tam = 0;
      while (p != l.cabeca) {
            tam++;
            p = p->prox;
      }
      return(tam);
}
```

```
// Posição da chave de busca na lista ordenada
NO* buscaSeqOrd(int ch, LISTA l, NO* *ant) {
      NO* p = l.cabeca->prox;
      *ant = l.cabeca;
      l.cabeca->chave = ch; // usa cabeça como sentinela
      while(p->chave < ch)    {
            *ant = p;
            p = p->prox;
      }
      if((p != l.cabeca) && (p->chave == ch) )  return(p);
      else return(NULL);
}

// Inserção ordenada sem duplicidade
bool inserirElemListaOrd(int ch , LISTA *l) {
      NO* novo;
      NO* ant;
      novo = buscaSeqOrd(ch, *l, &ant);
      if(novo) return(false);
      novo = (NO*) malloc(sizeof(NO));
      novo->chave = ch;
      novo->prox = ant->prox;
      ant->prox = novo;
      return(true);
}

// Destruição da lista curcular e com nó cabeça
void destruirLista (LISTA *l) {
      NO* atual;
      NO* prox;
      atual = l->cabeca->prox;
      while (atual != l->cabeca )   {
          prox = atual->prox;
          free(atual);
          atual = prox;
      }
      l->cabeca->prox = l->cabeca;
}
```

Listas dinâmicas duplamente encadeadas com nó cabeça e circularidade

Quando necessitamos percorrer a lista indistintamente em ambas as direções, usamos encadeamento duplo (ligando cada nó ao seu antecessor e ao seu sucessor).

```
typedef struct estrutura {
        int chave;
        int info;
        estrutura *prox;
        estrutura *ant;
} NO;

typedef struct {
        NO* cabeca;
} LISTA;
```

```
// Inicialização (encadeamento duplo, circular e com nó cabeça)
void inicializarLista(LISTA *l) {
     l->cabeca = (NO*) malloc(sizeof(NO));
     l->cabeca->prox = l->cabeca;
     l->cabeca->ant = l->cabeca;
}

// Último elemento da lista (encadeamento duplo, circular e com nó cabeça)
NO* ultimoElemLista(LISTA l) {
   NO* p = l.cabeca->ant;
   if(p == l.cabeca) return(NULL);
   else return(p);
}

// Inserção ordenada sem duplicações
bool inserirElemListaOrd(int ch , LISTA *l) {
    NO* novo;
    NO* ant;
    novo = buscaSeqOrd(ch, *l, &ant);
    if(novo) return(false);
    novo = (NO*) malloc(sizeof(NO));
    novo->chave = ch;
    novo->prox = ant->prox;
    novo->ant = ant;
    novo->prox->ant = novo;
    ant->prox = novo;
    return(true);
}
```

2.3. Filas

Filas são listas lineares com disciplina de acesso FIFO (first-in, first-out, ou, primeiro a entrar é o primeiro a sair). Sua principal aplicação é o armazenamento de dados em que é importante preservar a ordem FIFO de entradas e saídas.

O comportamento de fila é obtido armazenando-se a posição das extremidades da estrutura (chamadas aqui de fim e início), e permitindo entradas apenas na extremidade "fim" e retiradas apenas na extremidade "início".

A implementação pode ser estática (usando um vetor circular) ou dinâmica (com ponteiros) sem diferenças significativas em termos de eficiência, uma vez que estas operações só podem ocorrer nas extremidades da estrutura.

2.3.1. Implementação dinâmica

```
typedef struct estrutura {
        int chave;
        estrutura *prox;
} NO;

typedef struct {
        NO* inicio;
        NO* fim;
} Fdinam;

// Inicialização da fila dinamica
void inicializarFdinam(Fdinam *f) {
     f->inicio = NULL;
     f->fim = NULL;
}

// quantos elementos existem
int tamanhoFdinam(Fdinam f) {
    NO* p;
    int tam = 0;
    p = f.inicio;
      while (p)   {
                  tam++;
                  p = p->prox;
   }
   return(tam);
}
```

```
// inserir item ao final da fila dinamica
void entrarFdinam(int ch, Fdinam *f) {
     NO* novo;
     novo = (NO*) malloc(sizeof(NO));
     novo->chave = ch;
     novo->prox = NULL;
     if(f->fim) f->fim->prox = novo;      // fila não é vazia
     else f->inicio = novo;               // 1a. inserção em fila vazia
     f->fim = novo;
}

// retirar a chave da frente ou -1
int sairFdinam(Fdinam *f)
{
    NO* aux;
    int ch;
    if(!f->inicio) return(-1);
    ch = f->inicio->chave;
    aux = f->inicio;
    f->inicio = f->inicio->prox;
    free(aux);
    if(!f->inicio) f->fim = NULL;         // fila ficou vazia
    return(ch);
}
```

2.3.2. Implementação Estática

```
typedef struct  {
        int chave;
} RegistroEstat;

typedef struct  {
        int inicio;
        int fim;
        RegistroEstat A[MAX];
} Festat;


// Inicializacao da fila estática
void inicializarFestat(Festat *f) {
     f->inicio = -1;
     f->fim = -1;
}

// Inserir novo item ao final
bool entrarFestat(int ch, Festat *f) {
     if(tamanhoFestat(*f) >= MAX) return(false);
     f->fim = (f->fim + 1) % MAX;
     f->A[f->fim].chave = ch;
     if(f->inicio < 0 ) f->inicio = 0;
     return(true);
}
```

```
// Retirar um item da frente ou retornar -1 se vazia
int sairFestat(Festat *f) {
     if(f->inicio < 0) return(-1);
     int ch = f->A[f->inicio].chave;
     if(f->inicio != f->fim)
            f->inicio = (f->inicio + 1) % MAX;
     else {
            f->inicio = -1;
            f->fim = -1;
     }
     return(ch);
}
```

2.4. Deques (Filas de duas pontas – double-ended queues)

Deques são filas que permitem tanto entrada quanto retirada em ambas extremidades. Neste caso não faz mais sentido falar em início e fim de fila, mas simplesmente início1 e início2.

Implementações estáticas e dinâmicas são possíveis. A implementação dinâmica tira proveito do encadeamento duplo para permitir acesso a ambas extremidades em tempo O(1). A implementação estática faz uso de um vetor circular semelhante ao usado para implementar a fila estática comum.

```
typedef struct estrutura {
        int chave;
        estrutura *prox;
        estrutura *ant;
} NO;


typedef struct {
        NO* inicio1;
        NO* inicio2;
} DEQUE;

// Inicialização do deque
void inicializarDeque(DEQUE *d) {
     d->inicio1 = NULL;
     d->inicio2 = NULL;
}
```

```
// Quantos elementos existem
int tamanhoDeque(DEQUE d) {
      NO* p = d.inicio1;
      int tam = 0;
      while(p)    {
          tam++;
          p = p->prox;
      }
      return(tam);
}

// Inserir no inicio1 do deque
void entrarDeque1(int ch, DEQUE *d) {
     NO* novo = (NO*) malloc(sizeof(NO));
     novo->chave = ch;
     novo->ant = NULL;
     novo->prox = d->inicio1;
     if(d->inicio1)   d->inicio1->ant = novo;  // já contém dados
     else d->inicio2 = novo;                   // 1a. inserção
     d->inicio1 = novo;
}

// Retirar de inicio1 ou retornar -1 se vazio
int sairDeque1(DEQUE *d) {
    NO* aux;
    if(!d->inicio1) return(-1);
    aux = d->inicio1;
    int ch = aux->chave;
    d->inicio1 = d->inicio1->prox;
    free(aux);
    if(!d->inicio1) d->inicio2 = NULL;
    else d->inicio1->ant = NULL;
    return(ch);
}

// destruir deque dinâmico
void DestruirDeque(DEQUE *d) {
     while (d->inicio1) sairDeque1(d);
}
```

2.5. Pilhas

Pilhas são listas lineares com disciplina de acesso FILO (*first-in, last-out*, ou, o primeiro a entrar é o último a sair). Da mesma forma que as filas, sua principal aplicação é o armazenamento de dados em que é importante preservar a ordem (neste caso, FILO) de entradas e saídas.

A pilha armazena apenas a posição de uma de suas extremidades (chamada topo), que é o único local onde são realizadas todas as operações de entrada e saída. A operação de entrada de dados (sempre no topo da pilha) é chamada *push* e a retirada (também sempre do topo) é chamada *pop*.

A implementação pode ser estática (usando um vetor simples) ou dinâmica (com ponteiros) sem diferenças significativas em termos de eficiência, uma vez que a estrutura só admite estas operações no topo da estrutura.

2.5.1. Implementação dinâmica

```
typedef struct estrutura {
        int chave;
        estrutura *prox;
} NO;

typedef struct {
        NO* topo;
} Pdinam;


// Inicialização da pilha dinâmica
void inicializarPdinam(Pdinam *p) {
     p->topo = NULL;
}

// Quantos elementos existem
int tamanhoPdinam(Pdinam p) {
      NO* p1 = p.topo;
      int tam = 0;
      while(p1) {
            tam++;
            p1 = p1->prox;
      }
      return(tam);
}

// Inserir item no topo
void push(int ch, Pdinam *p) {
     NO* novo = (NO*) malloc(sizeof(NO));
     novo->chave = ch;
     novo->prox = p->topo;
     p->topo = novo;
}
```

```
// Retirar a chave do topo ou -1
int pop(Pdinam *p) {
    NO* aux;
    int ch;
    if(!p->topo) return(-1);
    aux = p->topo;
    ch = aux->chave;
    p->topo = p->topo->prox;
    free(aux);
    return(ch);
}
```

2.5.2. Implementação Estática

```
typedef struct {
        int chave;
} RegistroEstat;

typedef struct {
        int topo;
        RegistroEstat A[MAX];
} PESTAT;

// Inicialização da pilha estática
void inicializarPestat(Pestat *p) {
     p->topo = -1;
}

// A pilha estática está cheia ?
bool pilhaCheia(Pestat p) {
     if( p.topo >= MAX - 1 ) return(true);
     else return(false);
}

// Inserir no topo da pilha estática
bool push(int ch, Pestat *p) {
     if( tamanhoPestat(*p) >= MAX ) return(false);
     p->topo++;
     p->A[p->topo].chave = ch;
     return(true);
}

// Retirar do topo ou retornar -1 se vazia
int pop(Pestat *p) {
     if(p->topo < 0) return (-1);
     int ch = p->A[p->topo].chave;
     p->topo--;
     return(ch);
}
```

2.5.3. Representação de duas pilhas em um único vetor

Duas pilhas de implementação estática que não necessitam de toda sua capacidade simultaneamente podem ser representadas economicamente em um único vetor compartilhado.

As pilhas são posicionadas nas extremidades do vetor e crescem em direção ao centro. Supondo-se um vetor com posições indexadas de 0 até MAX-1, o topo da primeira pilha é inicializado com -1 e o topo da segunda é inicializado com MAX, correspondendo as situações de pilha vazia de cada estrutura. As duas pilhas estão simultaneamente cheias quando não há mais posições livres entre elas, ou seja, quando (topo2 - topo1 == 1).

```
typedef struct {
        int topo1;
        int topo2;
        int A[MAX];
} PILHADUPLA;

// Inicializacao da pilha dupla
void inicializarPilhaDupla(PILHADUPLA *p) {
          p->topo1 = -1;
          p->topo2 = MAX;
}

// Quantos elementos existem na pilha k (1 ou 2)
int tamanhoPilhaDupla(PILHADUPLA p, int k) {
     if( k == 1 ) return(p.topo1 + 1);
     else return(MAX - p.topo2);
}

// O vetor está cheio ?
bool vetorPilhaCheio(PILHADUPLA p) {
     if(p.topo1 == (p.topo2 - 1) ) return(true);
     else return(false);
}
```

```
// Inserir no topo da pilha k
bool pushK(int ch, PILHADUPLA *p, int k) {
      if(pilhaCheia(*p)) return(false);
      if(k == 1)  {
            p->topo1++;
            p->A[p->topo1] = ch;
      }
      else  {
            p->topo2--;
            p->A[p->topo2] = ch;
      }
      return(true);
}


// Retirar do topo k, ou retornar -1
int popK(PILHADUPLA *p, int k) {
      int ch = -1;
      if(k == 1)  {
            if(p->topo1 > -1) {
                  ch = p->A[p->topo1];
                  p->topo1--;
            }
      }
      else   {
            if(p->topo2 < MAX) {
                  ch = p->A[p->topo2];
                  p->topo2++;
            }
      }
      return(ch);
}
```

2.5.4. Representação de 'NP' pilhas em um único vetor

O caso geral de representação estática de 'NP' pilhas compartilhando um único vetor envolve o controle individual do topo de cada pilha e também da sua base. Cada topo [k] aponta para o último elemento efetivo de cada pilha (i.e., da mesma forma que o topo de uma pilha comum) mas, para que seja possível diferenciar uma pilha vazia de uma pilha unitária, cada base [k] aponta para o elemento anterior ao primeiro elemento real da respectiva pilha.

Uma pilha k está vazia se base[k] == topo[k]. Uma pilha [k] está cheia se topo[k] == base[k+1], significando que a pilha [k] não pode mais crescer sem sobrepor-se a pilha [k+1].

```
# define MAX 15   // tamanho do vetor A
# define NP 5     // nro. de pilhas compartilhando o vetor (numeradas de 0..NP-1)
```

A especificação da estrutura inclui as NP pilhas reais (numeradas de 0 até NP-1) e mais uma pilha 'extra' (fictícia) de índice NP cuja função é descrita a seguir.

```
typedef struct {
        int base[NP+1]; // pilhas [0..NP-1] + pilha[NP] auxiliar
        int topo[NP+1];
        int A[MAX];
} PILHAS;
```

Na inicialização da estrutura, cada topo é igualado a sua respectiva base, o que define uma pilha vazia. Além disso, para evitar acúmulo de pilhas em um mesmo ponto do vetor (o que ocasionaria grande movimentação de dados nas primeiras entradas de dados), todas as NP+1 pilhas são inicializadas com suas bases distribuídas a intervalos regulares ao longo da estrutura.

A pilha[0] fica na posição –1. A pilha[NP] - que é apenas um artifício de programação - fica permanentemente na posição MAX-1, servindo apenas como marcador de fim do vetor (i.e., esta pilha jamais será afetada pelas rotinas de deslocamento). A pilha extra será sempre vazia e será usada para simplificar o teste de pilha cheia, que pode se basear sempre na comparação entre um topo e a base da pilha seguinte, ao invés de se preocupar também com o fim do vetor.

```
// Inicializacao da pilha múltipla
void inicializarPilhas(PILHAS *p) {
      int i;
      for(i = 0; i <= NP ; i++) {
            p->base[i] = ( i * (MAX / NP) ) - 1;
            p->topo[i] = p->base[i];
      }
}

// Quantos elementos existem na pilha k
int tamanhoPilhaK(PILHAS p, int k) {
     return(p.topo[k] - p.base[k]);
}
```

A utilidade da pilha ‘fictícia’ de índice [NP] fica claro na rotina a seguir, na qual não é preciso se preocupar com o caso especial em que k é a última pilha real (i.e., quando k == NP-1).

```
// A pilha k esta cheia ?
bool pilhaKcheia(PILHAS p, int k) {
     if(p.topo[k] == p.base[k + 1])
            return(true);
     else
            return(false);
}
```

```
// Desloca pilha k uma posição para a direita, se possivel
bool paraDireita(PILHAS *p, int k) {
      int i;
      if( (k < 1) || (k > NP-1) ) return (false); // índice inválido
      if( (p->topo[k] < p->base[k + 1])) {
            for(i = p->topo[k] + 1; i > p->base[k]; i--) p->A[i] = p->A[i-1];
            p->topo[k]++;
            p->base[k]++;
            return(true);
      }
      return(false);
}
```

A inclusão de um novo elemento no topo de uma pilha k deve considerar a existência de espaço livre e, se for o caso, movimentar as estruturas vizinhas para obtê-lo. No exemplo a seguir, o programa tenta deslocar todas as pilhas à direita de k em uma posição para a direita. Se isso falhar, tenta deslocar todas as pilhas da esquerda (inclusive k) uma posição para a esquerda. Se isso ainda não resultar em uma posição livre no topo de k, então o vetor está totalmente ocupado e a inserção não pode ser realizada.

```
// Inserir um novo item no topo da pilha k
bool pushK(int ch, PILHAS *p, int k) {
     int j;
     if( (pilhaKcheia(*p, k)) && (k < NP-1) )
        // desloca p/direita todas as pilhas de [k+1..NP-1] em ordem reversa
        for( j = NP-1; j > k; j--) paraDireita(p, j);
     if( (pilhaKcheia(*p, k)) && (k > 0))
        // desloca p/esquerda todas as pilhas de [1..k] (mas não a pilha 0)
        for( j = 1; j <= k; j++) paraEsquerda(p, j);
     if(pilhaKcheia(*p, k)) return(false);
     p->topo[k]++;
     p->A[p->topo[k]] = ch;
     return(true);
}
```

Observe que este é apenas um exemplo de estratégia possível. Um procedimento talvez mais eficiente poderia tentar primeiro deslocar apenas a pilha k+1 para direita. Apenas quando isso falhasse poderia então tentar deslocar apenas as pilha k-1 e a pilha k para a esquerda, e só em caso de última necessidade tentaria deslocaria várias pilhas simultaneamente como feito acima.

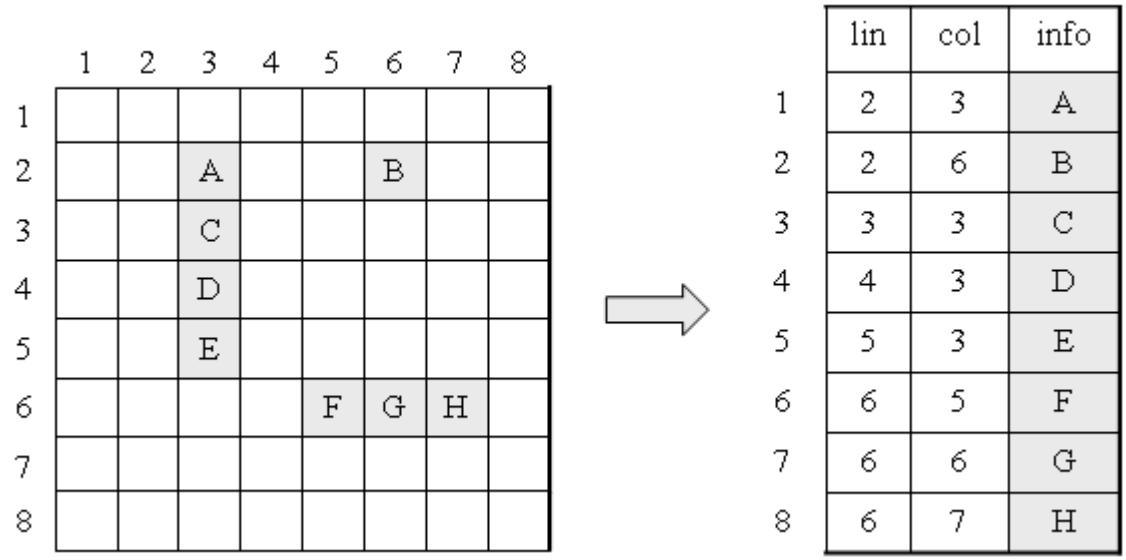
```
// Retirar um item da pilha k, ou -1
int popK(PILHAS *p, int k) {
      int resp = -1;
      if( (p->topo[k] > p->base[k]) )  {
            resp = p->A[p->topo[k]];
            p->topo[k]--;
      }
      return(resp);
}
```

2.6. Matrizes Esparsas

Uma matriz esparsa é uma matriz extensa na qual poucos elementos são não-nulos (ou de valor diferente de zero). O problema de representação destas estruturas consiste em economizar memória armazenando apenas os dados válidos (i.e., não nulos) sem perda das propriedades matriciais (i.e., a noção de posição em linha e coluna de cada elemento). As implementações mais comuns são a representação em linhas ou por listas cruzadas.

2.6.1. Representação por Linhas

A forma mais simples (porém não necessariamente mais eficiente) de armazenar uma matriz esparsa é na forma de uma tabela (geralmente, uma lista ligada) de nós contendo a linha e coluna de cada elemento e suas demais informações. A lista é ordenada por linhas para facilitar o percurso neste sentido. Já que nem todos os elementos são representados, certas colunas e linhas podem não existir na lista ligada. Por este motivo, constantes *maxlin* e *maxcol* são usadas para armazenar as dimensões reais da matriz, seja através de declarações globais ou dentro do próprio tipo MATRIZ.



	lin	col	info
1	2	3	A
2	2	6	B
3	3	3	C
4	4	3	D
5	5	3	E
6	6	5	F
7	6	6	G
8	6	7	H

A matriz será acessível a partir do ponteiro de início da lista ligada que a representa.

```
typedef struct estrutura {
      int lin;
      int col;
      TIPOINFO info;
      estrutura *prox;
} NO;

typedef struct {
      NO* inicio;
      int maxlin;
      int maxcol;
} MATRIZ;
```

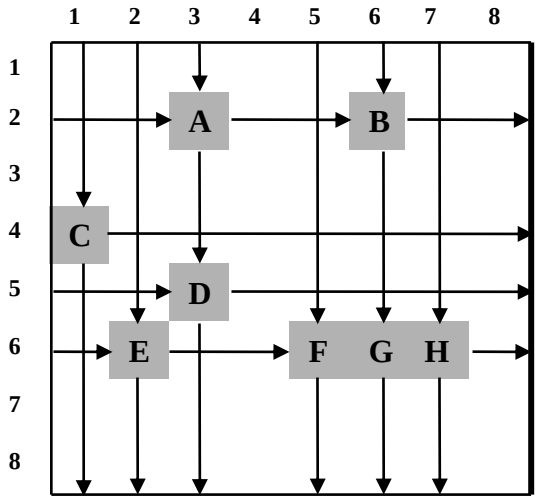
A economia de espaço desta representação é bastante significativa. Para uma matriz de *maxlin* x *maxcol* elementos, dos quais n são não-nulos, seu uso será vantajoso (do ponto de vista da complexidade de espaço) sempre que:

```
(maxlin*maxcol*sizeof(TIPOINFO)) > (n*(sizeof(NO))
Lembrando que um nó é composto de dois inteiros, um TIPOINFO e um NO*.
```

Por outro lado, a representação por linhas não apresenta bom *tempo de resposta* para certas operações matriciais. Em especial, percorrer uma linha da matriz exige que todas as linhas acima dela sejam percorridas. Pior do que isso, percorrer uma coluna da matriz exige que a matriz inteira (ou melhor, todos os seus elementos não-nulos) seja percorrida. Considerando-se que matrizes esparsas tendem a ser estruturas de grande porte, em muitas aplicações estes tempos de execução podem ser inaceitáveis.

2.6.2. Representação por Listas Cruzadas

Com um gasto adicional de espaço de armazenamento, podemos representar uma matriz esparsa com tempo de acesso proporcional ao volume de dados de cada linha ou coluna. Nesta representação, usamos uma lista ligada para cada linha e outra para cada coluna da matriz. As listas se cruzam, isto é, compartilham os mesmos nós em cada intersecção, e por este motivo cada nó armazena um ponteiro para a próxima linha (abaixo) e próxima coluna (à direita).




```
typedef struct estrutura {
      int lin;
      int col;
      TIPOINFO info;
      estrutura *proxL;
      estrutura *proxC;
} NO;
```

O acesso a cada linha ou coluna é indexado por meio de um vetor de ponteiros de linhas e outro de colunas. Cada ponteiro destes vetores indica o início de uma das listas da matriz. As dimensões MAXLIn e MAXCOL foram aqui tratadas como constantes globais para facilitar a definição dos vetores.

```
typedef struct {
      NO* lin[MAXLIN+1]; // para indexar até MAXLIN
      NO* col[MAXCOL+1]; // para indexar até MAXCOL
} LISTASCR;
```

Para uma matriz de MAXLIN x MAXCOL elementos, dos quais n são não-nulos, a representação por listas cruzadas será vantajosa (do ponto de vista da complexidade de espaço) sempre que:

```
(MAXCOL*MAXLIN*sizeof(TIPOINFO)) > {n*(sizeof(NO))+sizeof(NO*)*(MAXLIN+MAXCOL)}
Lembrando que um nó é composto de dois inteiros, um TIPOINFO e dois ponteiros.

// Inicialização
void inicializarMatriz(LISTASCR *m) {
      int i;
      for(i=1; i <= MAXLIN; i++)
            m->lin[i] = NULL;
      for(i=1; i <= MAXCOL; i++)
            m->col[i] = NULL;
     }
}

// Conta elementos da estrutura
int contaElementos(LISTASCR m) {
      int i, t;
      NO* p;
      t = 0;
      for(i=1; i<= MAX; i++) {
            p = m.lin[i];
            while (p) {
                  t++;
                  p = p->proxC;
            }
      }
      return(t);
}
```

```
/ Exclui elemento p das listas apontadas por *linha e *coluna
void excluir(NO* *linha, NO* *coluna, NO* esq, NO* acima, NO* p) {
      if(p)  {
             // desliga do vetor de linhas
             if(esq)
                esq->proxC = p->proxC;
             else
                 *linha = p->proxC;
             // desliga do vetor de colunas
             if(acima)
                acima->proxL = p->proxL;
             else
                 *coluna = p->proxL;
             free(p);
        }
}

// Exibe os produtos dos elementos das colunas c1 e c2
void exibirProdutos(LISTASCR m, int c1, int c2) {
     NO* p1 = m.col[c1];
     NO* p2 = m.col[c2];
     int atual = 0;     // linha a ser processada
     int v1, v2;
     while ((p1)||(p2)) {
           v1 = 0;
           if(p1) {
                 if(p1->lin < atual) {
                    p1 = p1->proxL;
                    continue;
                 }
                 if(p1->lin == atual)
                    v1 = p1->chave;
           }
           v2 = 0;
            if(p2)
           {
                 if(p2->lin < atual) {
                    p2 = p2->proxL;
                    continue;
                 }
                 if(p2->lin == atual)
                    v2 = p2->chave;
           }
           printf("Linha %d*%d=%d\n", v1, v2, v1*v2);
           atual++;
     }
     if(atual < MAXLIN+1) for(; atual <= MAXLIN; atual++)
            printf("Linha %d*%d=%d\n", atual,0,0);
}
```

2.7. Listas Generalizadas

São listas contendo dois tipos de elementos: elementos ditos “normais” (e.g., que armazenam chaves) e elementos que representam entradas para sublistas. Para decidir se um nó armazena uma chave ou um ponteiro de sublista, usamos um campo *int tipo* cuja manutenção é responsabilidade do programador. Dependendo do valor de *tipo*, armazenamos em um campo de tipo variável (um *union* em C) o dado correspondente. Por exemplo, podemos convencionar que *tipo*=1 representará elementos regulares da lista (i.e., contendo chaves), e que *tipo*=2 representará uma sublista (i.e., contendo um ponteiro para esta).

```
typedef struct estrutura {
        int tipo; // 1=elemento e 2=sublista
        union {
              int chave;
              struct estrutura *sublista;
        };
        estrutura *prox;
} NO;

// Inicialização
void inicializarLista(NO* *p) {
    *p = NULL;
}

// Quantidade de chaves na lista
int contarChaves(NO* p) {
      int chaves = 0;
      while (p)   {
            if( p->tipo == 1) // elemento regular
               chaves++;
            else
               chaves = chaves + contarChaves(p->sublista);
            p = p->prox;
    }
    return(chaves);
}
```

```
// Quantidade total de nós na lista
int contarNos(NO* p) {
      int nos = 0;
      while (p)   {
            nos++;
            if( p->tipo == 2) // sublista
               nos = nos + ContarNos(p->sublista);
            p = p->prox;
    }
    return(nos);
}

// Profundidade maxima da lista
int profundidade(NO* p) {
      int maximo = 0;
      int resp;
      if(!p) return(maximo);
      while(p) {
              if( p->tipo == 1) resp = 0;
              else resp = profundidade(p->sublista);
              if(resp > maximo) maximo = resp;
              p = p->prox;
      }
      return(maximo + 1);
}

// copia uma lista inteira
NO* copiarListaGen(NO* p) {
      NO* novo;
      NO* abaixo;
      NO* dir;
      int tipo;
      novo = NULL;
      if (p) {
            tipo = p->tipo;
            if( tipo == 2)
                 abaixo = copiarListaGen(p->sublista);
            dir = copiarListaGen(p->prox);
            novo = (NO *) malloc(sizeof(NO));
            novo->tipo = tipo;
            novo->prox = dir;
            if( tipo == 1)
                novo->chave = p->chave;
            else
                novo->sublista = abaixo;
      }
      return(novo);
}
```

```
// verifica se duas listas são identicas
bool listasIguais(NO* a, NO* b) {
      bool resp = false;
      if((!a) && (!b)) return(true);
      if((a) && (b)) {
            if( a->tipo == b->tipo) {
                  if( a->tipo == 1)
                        resp = (a->chave == b->chave);
                  else
                        resp = listasIguais(a->sublista, b->sublista);
                  if(resp)
                        resp = listasIguais(a->prox, b->prox);
            }
      }
      return(resp);
}
```

3. Listas Não Lineares

Quando existe mais de um caminho possíveis pela estrutura, esta é dita não linear. Exemplos clássicos de estruturas deste tipo são as árvores e grafos (estes estudados em Algoritmos e Estruturas de Dados II).

3.1. Árvores

Uma árvore é um conjunto de nós composto de um nó especial (chamado raiz) e conjuntos disjuntos de nós subordinados ao nó raiz que são eles próprios (sub)árvores.

Terminologia

Grau de um nó: a quantidade de subárvores do nó;
Grau de uma árvore: grau máximo dentre todos os nós da estrutura;
Folhas de uma árvore: nós de grau zero;
Filhos de x: raízes das subárvores de *x*; *x* é o nó pai de seus filhos;
Ancestrais de x: todos nós no caminho desde a raiz até *x*.
Nível de x: a raiz é nível 1; se um nó está no nível *n*, seus filhos estão no nível *n*+1;
Altura de um nó folha é sempre 1; *(convenção adotada nesta apostila – outros autores consideram zero)*
Altura de um nó não folha: a altura máxima dentre todas suas subárvores + 1;
Altura de uma árvore é a altura de sua raiz.

Uma árvore de grau *m* é dita *m*-ária. Árvores *m*-árias são de difícil representação e manipulação (por exemplo, a definição de muitos ponteiros em cada nó representa um grande desperdício de espaço ocupado por ponteiros NULL). Por este motivo, árvores *m*-árias são geralmente representadas por árvores binárias (veja definição a seguir) sem perda de propriedades.

Em computação, árvores (e especialmente árvores binárias) são usadas para armazenar dados (chaves e outros campos de informação) em seus nós da mesma forma que listas sequenciais e listas ligadas.

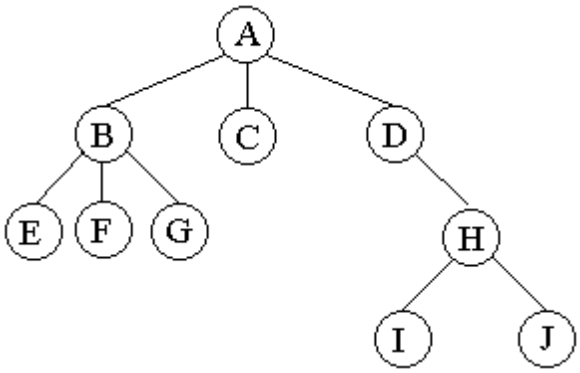
3.1.1. Árvores Binárias

Uma árvore binária é uma estrutura vazia ou um nó raiz e duas subárvores chamadas esquerda e direita, as quais são também árvores binárias (vazias ou não). É importante observar que uma árvore binária não é apenas uma árvore de grau máximo dois, pois há também a questão de ordem (esquerda e direita) de subárvores, conceito este que não existe na definição de árvore comum discutida no item anterior.

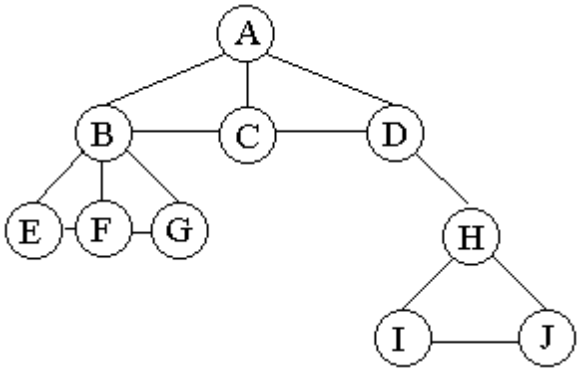
Uma vez que árvores *m*-árias não definem uma ordem específica entre as subárvores de cada nó, a conversão de árvore *m*-ária para binária é trivial: basta eleger um filho qualquer como raiz da subárvore esquerda, e os demais como subárvore direita e seus descendentes. O procedimento de conversão compreende dois passos:

(a) todos os nós irmãos da árvore *m*-ária são interligados;
(b) todas as conexões pai-filho são removidas, exceto a primeira de cada grupo.

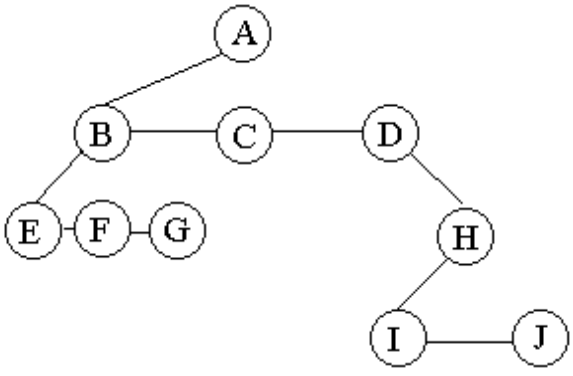
A árvore assim resultante pode ser redesenhada na posição correta (preservando-se as ligações esquerda e direita estabelecidas) formando uma árvore binária.



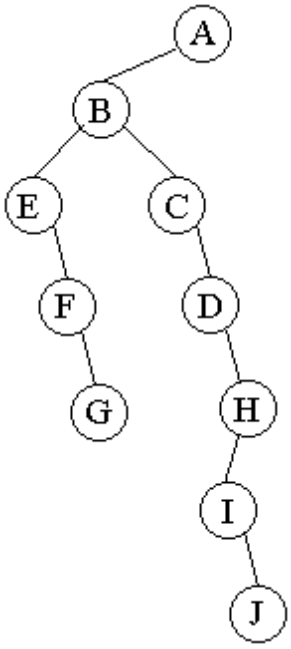
Árvore m-ária a ser convertida



Passo (a) nós irmãos são interligados



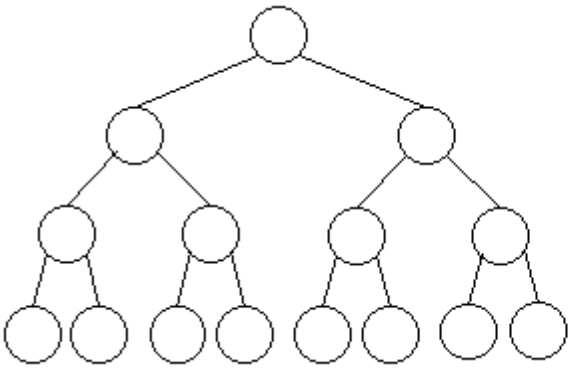
Passo (b): ligações pai-filho são removidas, exceto a primeira de cada grupo



Árvore binária resultante (as ligações pai-filho originais são mantidas)

Propriedades

- O número máximo de nós possíveis no nível *i* é 2^{i-1}



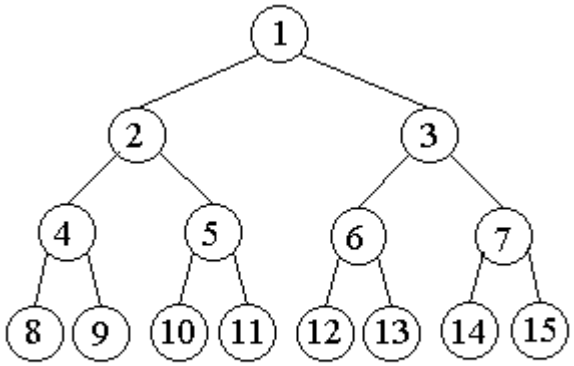
nível $1 = 2^{1-1} = 1$ nó

nível $2 = 2^{2-1} = 2$ nós

nível $3 = 2^{3-1} = 4$ nós

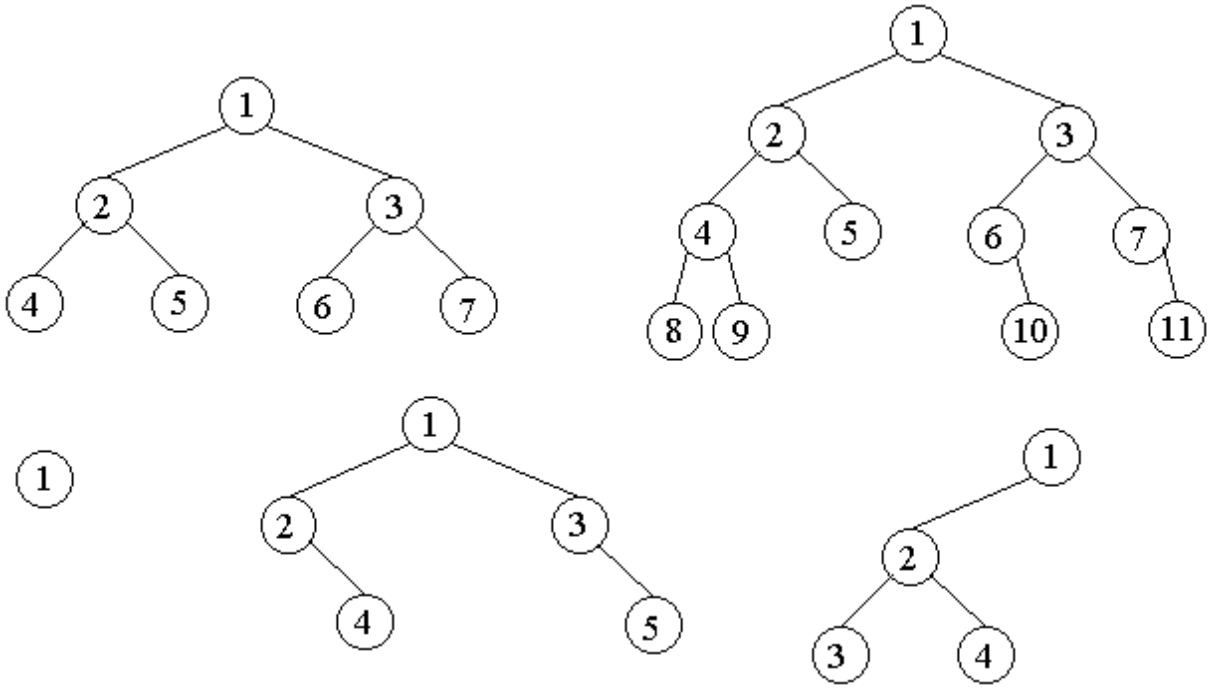
nível $4 = 2^{4-1} = 8$ nós

- Uma árvore binária de altura *h* tem no máximo $2^h - 1$ nós.



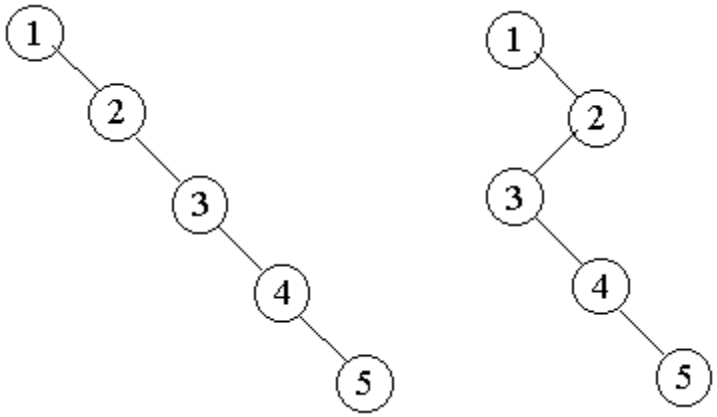
para $h = 4$, $n_{máx} = 2^4 - 1 = 15$

- A altura mínima de uma árvore binária com $n > 0$ nós é $1 + \text{chão}(\log_2 n)$
- Uma árvore binária de altura mínima é dita ***completa***.



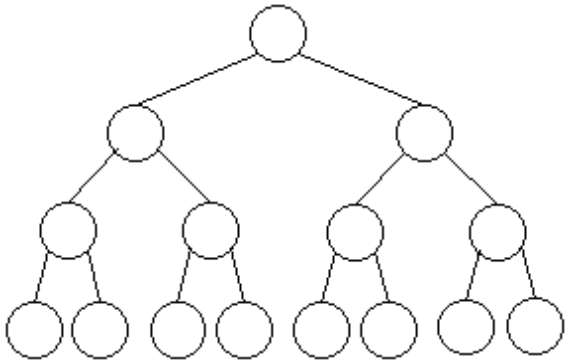
Exemplos de árvores completas. Seus nós não podem ser redistribuídos formando uma árvore de altura menor do que estas.

- Uma árvore binária de altura máxima para uma quantidade de n nós é dita ***assimétrica***. Neste caso, a altura é $h = n$ e seus nós interiores possuem exatamente uma subárvore vazia cada.



Exemplos de árvores assimétricas

- Uma árvore binária de altura h é ***cheia*** se possui exatamente $2^h - 1$ nós.



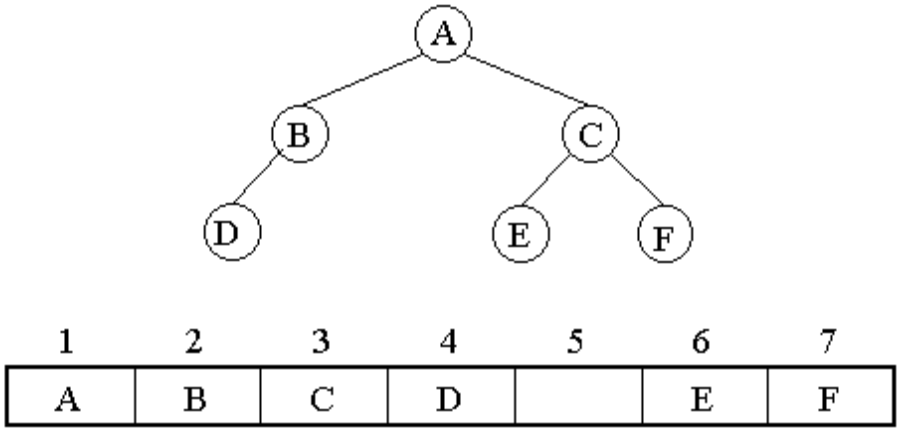
Exemplo de árvore cheia – possuindo o número máximo de nós para a sua altura.

Implementação Estática

Embora a implementação dinâmica seja mais comum, árvores binárias podem ser representadas em um vetor. Isso é especialmente útil em aplicações que não sofrem grande volume de inserções e exclusões; nos demais casos a representação dinâmica continua sendo a preferida.

Na representação estática, o vetor deve ser grande o suficiente para conter o número máximo possível de nós para a altura máxima h_{max} estabelecida, ou seja, deve ter no mínimo $2^{hmax} - 1$ posições. Isso é necessário porque mesmo os nós não existentes terão seu espaço reservado no vetor, já que a posição relativa de qualquer pai ou filho no vetor é fixa, definida em função das posições de seus antecessores.

Os nós da árvore são dispostos ao longo do vetor em níveis, da esquerda para a direita, começando pela raiz (que ocupa a primeira posição). Como os nós inexistentes deixam posições vazias no vetor, algum tipo de controle deve ser feito para diferenciar posições livres e ocupadas (e.g., com uso de um campo booleano).



Uma vez que a raiz ocupa a primeira posição do vetor, os outros nós têm suas posições definidas como segue:

- pai(*i*) = chão (*i* / 2) se *i* == 1, não há pai
- filho_esq(*i*) = 2 * *i* se *i* > *n*, não há filho esquerdo
- filho_dir(*i*) = (2 * i) + 1 se *i* > *n*, não há filho direito

Implementação Dinâmica

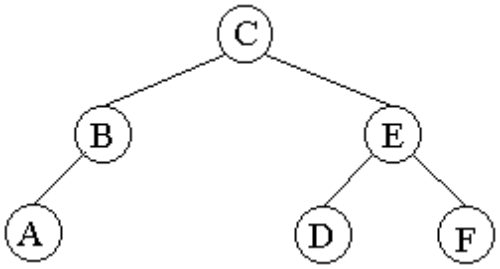
```
typedef struct estrutura {
        int chave;
        estrutura *esq;
        estrutura *dir;
} NO;

// Inicialização da árvore vazia
void inicializarArvore(NO* *raiz) {
     *raiz = NULL;
}
```

```
// Verificar se árvore é vazia
bool arvoreVazia(NO* raiz) {
        if(!raiz) return(true);
        else return(false);
}

// Inserção de um nó em árvore comum (sem ordem) esq:pos=1 dir:pos=2

bool inserirNo(NO* *raiz, NO* pai, int ch, int pos) {
        NO* novo;
        if(pai) {
                if(     ((pos==1) && (pai->esq!=NULL))||
                        ((pos==2) && (pai->dir!=NULL)))   {
                        return(false);
                }
        }
        novo = (NO *) malloc(sizeof(NO));
        novo->chave = ch;
        novo->esq = NULL;
        novo->dir = NULL;
        if(!pai) *raiz = novo;
        else  {
          if(pos==1) // esquerda
                pai->esq = novo;
          else
                pai->dir = novo;
        }
}
```

Percursos em árvore binária

Convenção: as operações possíveis (determinadas pelos ponteiros existentes) são três: visitar a raiz, deslocar-se para a esquerda e deslocar-se para a direita. A esquerda sempre tem prioridade sobre direita.

Os percursos possíveis de acordo com esta convenção são:
- *Pré-ordem*: visita a raiz, esquerda e direita.
- *Em ordem*: esquerda, visita a raiz e direita.
- *Pós-ordem*: esquerda, direita e visita raiz.



Pré-ordem:	C B A E D F
Em-ordem:	A B C D E F
Pós-ordem:	A B D F E C

Algoritmos não-recursivos

Os percursos básicos em árvore binária (em especial, o de pré-ordem e o em ordem) podem ser obtidos com uma implementação não recursiva usando uma estrutura auxiliar do tipo pilha.

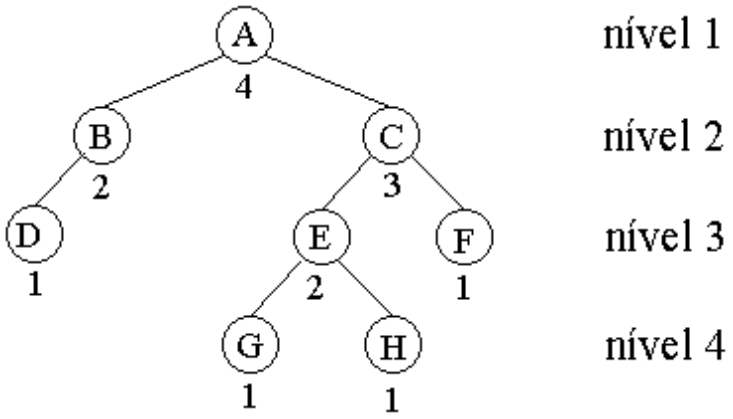
```
// Percurso pré-ordem não-recursivo (raiz, esquerda, direita)
void preOrdemNaoRecurs(NO* p) {
      PILHA pi;
      inicializarPilha(&pi); // deve ser uma pilha de ponteiros de nós da árvore

      while (true) {
            while (p) {
                  visita(p);
                  if(p->dir) push(p->dir, &pi); // memoriza caminho à direita
                  p = p->esq;
            }
            if(tamanhoPilha(pi) > 0)
                p = pop(&pi);
            else break;
      }
}
```

Algoritmos recursivos

Os percursos básicos em árvore binária são facilmente obtidos com uma implementação recursiva.

```
void preOrdem(NO* p)
{
        if(p)   {
                visita(p);
                preOrdem(p->esq);
                preOrdem(p->dir);
        }
}

void emOrdem(NO* p)
{
        if(p)   {
                emOrdem(p->esq);
                visita(p);
                emOrdem(p->dir);
        }
 }

void posOrdem(NO* p)
{
        if(p)   {
                posOrdem(p->esq);
                posOrdem(p->dir);
                visita(p);
        }
}
```

Outros percursos



Em altura: D G H F B E C A
Em nível: A B C D E F G H

Os algoritmos para estes percursos não são recursivos. O percurso em nível, por exemplo, é mais facilmente obtido com uma estrutura auxiliar do tipo fila, usada para armazenar os filhos de cada nó visitado e então visitá-los na ordem em que são retirados da mesma.

```
// Percorre a árvore em nível e exibe
void exibirArvoreEmNivel(NO* raiz) {
      FILA f;
      NO* p = raiz;
      inicializarFila(&f);
      while( (p) || (f.inicio) )    {
               if(p->esq) entrarFila(p->esq, &f);
               if(p->dir) entrarFila(p->dir, &f);
               printf("%d", p->chave);
               p = NULL;
               if(f.inicio) p = sairFila(&f);
     }
}
```

Algoritmos em Árvore Binária Comum

```
// Procura a chave na arvore inteira usando pré-ordem (usando retorno NO*)
NO* busca_local(NO* p, int ch, bool *achou) {
      NO* aux;
      *achou = false;
      if(!p)
            return(NULL);
      if(p->chave == ch) {
            *achou = true;
            return(p);
      }
      aux = busca_local(p->esq, ch, achou);
      if(*achou)
            return(aux);
      else
            return(busca_local(p->dir, ch, achou));
}

NO* buscaCompleta(NO* raiz, int ch) {
    bool achou;
    return(busca_local(raiz, ch, &achou));
}


// Destruir uma árvore usando pós-ordem
void destruirArvore(NO* *p) {
      if(*p) {
            destruirArvore(&(*p)->esq);
            destruirArvore(&(*p)->dir);
            free(*p);
      }
      *p = NULL;  // evita que a raiz aponte para endereço inválido
}


// Retorna o endereço do pai da chave procurada (usando função void)
void travessiap(NO* p, int ch, NO* *no_pai, bool *achou)   {
      if(p) {
            if(p->esq)
                  if(p->esq->chave == ch) {
                        *achou = true;
                        *no_pai = p;
                  }
            if(!*achou)
                  if(p->dir)
                        if(p->dir->chave == ch)  {
                              *achou = true;
                              *no_pai = p;
                        }
            if(!*achou)
                  travessiap(p->esq, ch, no_pai, achou);
            if(!*achou)
                  travessiap(p->dir, ch, no_pai, achou);
        }
}
```

```
NO* pai(NO* raiz, int ch) {
      bool achou = false;
      NO* no_pai;
      if(raiz)  {
            if(raiz->chave == ch)
                  return(NULL);
            else {
                  travessiap(raiz, ch, &no_pai, &achou);
                  return(no_pai);
            }
      }
}


// Retorna o nível de uma chave (que deve ser encontrada)
void travessia(NO* p, int *niv, int ch, bool *achou) {
      if(p)  {
            *niv = *niv + 1;
            if(p->chave == ch)
                  *achou = true;
            else   {
                  travessia(p->esq, niv, ch, achou);
                  if((!*achou))
                        travessia(p->dir, niv, ch, achou);
                  if((!*achou))
                        *niv = *niv - 1;
            }
       }
}

int nivel (NO* raiz, int ch) {
      int n = 0;
      bool achou = false;
      travessia(raiz, &n, ch, &achou);
      return(n);
}
```

Árvores de Busca Binária (ABB)

Da mesma forma que no caso das listas lineares de implementação estática e dinâmica, talvez a aplicação mais importante de árvores binárias seja seu uso como *tabelas* para armazenamento de dados de forma eficiente. Para este propósito, os dados contidos na estrutura são ordenados de forma a viabilizar a busca binária de suas chaves.

Uma árvore de busca binária é uma árvore binária em que todos nós apresentam a seguinte propriedade: dado um nó p da estrutura, todos os descendentes esquerdos de p têm um valor de chave menor do que a chave de p, e todos os descendentes direitos de p têm um valor de chave maior do que a chave de p. A estrutura obviamente não pode armazenar chaves repetidas, e seu formato depende inteiramente da ordem em que as chaves são inseridas. Por este motivo, quanto menor a altura de uma árvore de busca, melhor o seu desempenho, pois o caminho para encontrar uma chave qualquer será o mais curto possível.

A possibilidade de operar busca binária (em tempo logarítmico, como fazemos em um vetor) aliada às vantagens da implementação dinâmica (i.e., inserção e exclusão em tempo constante) são as principais razões do grande êxito das árvores de busca binária como estruturas de armazenamento de tabelas de chaves, e, consequentemente, do seu uso generalizado na computação.

```
// Busca binária não recursiva devolvendo o nó pai
NO* buscaNo(NO* raiz, int ch, NO* *pai) {
      NO* atual = raiz;
      *pai = NULL;
      while (atual)     {
            if(atual->chave == ch)
                  return(atual);
            *pai = atual;
            if(ch < atual->chave)
                  atual = atual->esq;
            else
                  atual = atual->dir;
      }
      return(NULL);
}
```

Na **exclusão** de uma chave em uma árvore de busca binária há 3 casos a tratar:

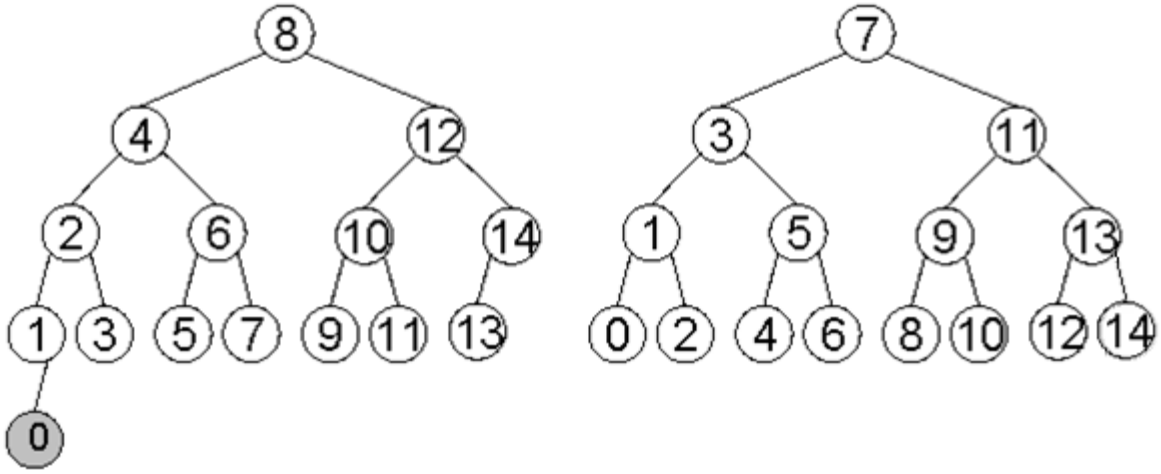
1. o nó a excluir não possui filhos; neste caso basta desalocar a memória e atualizar o ponteiro do nó pai.
2. o nó a excluir possui apenas um filho (direito ou esquerdo): neste caso o filho passa a ocupar a posição do nó excluído.
3. o nó a excluir possui os dois filhos: neste caso a chave a ser excluída é substituída pela chave do maior descendente esquerda, ou pela chave do menor descendente direito do nó em questão. Depois disso, este nó descendente cuja chave for promovida à posição do nó excluído é excluído segundo o caso 1 ou 2 acima, já que ele próprio pode ter no máximo um filho (se tivesse dois filhos não poderia ser o maior descendente esquerdo ou menor descendente direito).

Uso de sentinela: para reduzir o número de comparações na busca, é possível criar (no momento da inicialização da árvore) um nó sentinela para o qual todos os ponteiros NULL da estrutura são direcionados. No momento da busca, a chave em questão é colocada no nó sentinela e assim não se faz necessário testar a condição de fim da estrutura, pois a chave será sempre encontrada (ou na sua posição real ou no sentinela).

```
// Inicializacao - sentinela
void inicializar(NO* *raiz, NO* sentinela) {
     *raiz = sentinela;
}

// Testa se árvore com sentinela é vazia
bool vazia(NO* raiz, NO* sentinela) {
   if (raiz == sentinela)
      return(true);
   else
      return(false);
}

// Busca binária com sentinela devolvendo o nó pai
NO* buscaComSentinela(NO* raiz, NO* sentinela, int ch, NO* *pai) {
      NO* atual = raiz;
      *pai = NULL;
      sentinela->chave = ch;
      while (atual->chave != ch)    {
            *pai = atual;
            if(ch < atual->chave)
                  atual = atual->esq;
            else
                  atual = atual->dir;
      }
      if(atual == sentinela)
            return(NULL);
      else
            return(atual);
}
```

A eficiência da busca em uma ABB com ou sem sentinela é determinada pela altura da estrutura. Uma árvore completa (de altura mínima da ordem lg n) permite uma busca de pior caso O(lg n), ou seja, tão eficiente quanto à busca binária em vetor. Por outro lado, uma árvore assimétrica (de altura máxima da ordem n) exige uma busca de pior caso O(n), ou seja, o mesmo que em uma lista ligada.

Árvores completas são entretanto difíceis de manter, ou seja, garantir que uma ABB continue sendo completa depois de cada inserção ou exclusão pode exigir a movimentação de todos os nós da estrutura em tempo O(n), ou seja, o mesmo problema que era verificado em listas sequenciais (vetores). No exemplo a seguir a inserção da chave 0 demonstra porque restaurar a condição de árvore completa é inviável na prática.



Inserção da chave 0 (à esquerda) e reorganização de todos os n nós
para manter árvore completa (à direita).

3.1.4. Árvores Balanceadas em Altura (AVL)

Ao invés de tentar manter uma árvore completa ao custo O(n), a solução para manter a busca eficiente é tentar algo mais simples, exigindo que a árvore seja apenas *balanceada*. Uma ABB balanceada é aquela que, embora não tendo necessariamente altura mínima, mantém a complexidade de busca de ordem logarítmica. Embora a busca em árvore balanceada nem sempre seja tão eficiente quanto seria em uma árvore completa, a reorganização necessária para manter uma árvore balanceada é localizada, e muito mais rápida do que a reorganização O(n) da árvore completa, em geral também de ordem logarítmica.

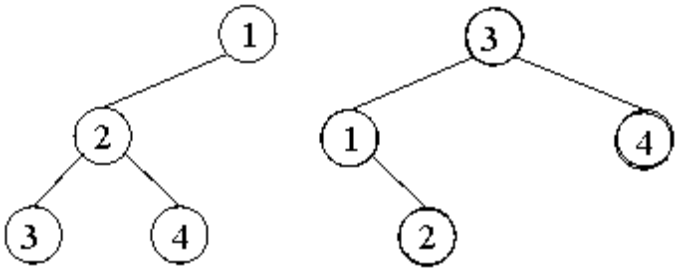
Existem muitos tipos de árvores balanceadas. Dentre as mais difundidas, e que serviram de base para vários outros modelos, estão as árvores balanceadas em altura, ou árvores AVL (dos criadores Adelson-Velskeii & Landis) propostas em 1962.

Uma árvore é dita AVL se todos os seus nós são AVL. Um nó é dito AVL se as alturas de sua subárvore direita (h_D) e esquerda ($h_{E)}$ não diferem em mais de uma unidade em módulo, ou seja:

$$| h_D - h_E | <= 1$$

A diferença entre a altura da subárvore direita e esquerda é chamada fator de balanceamento do nó AVL. Como o custo de cálculo deste fator é computacionalmente elevado, em geral o nó possui um campo para armazenar este tipo de informação, a qual deve ser atualizada pelos algoritmos de atualização da estrutura.

```
typedef struct estrutura {
        int chave;
        estrutura *esq;
        estrutura *dir;
        int bal;        // fator de balanceamento (0, -1 ou +1)
} NO;
```



Duas árvores completas, porém só a da direita é AVL.

```
// Verifica se árvore é AVL
bool ehavl(NO* p) {
      int e,d;
      bool ok = true;
      if(p)  {
            ok = ehavl(p->esq);
            if(ok) ok = ehavl(p->dir);
            if(ok) {
                  e = altura(p->esq);
                  d = altura(p->dir);
                  if(abs(e-d) <= 1)
                        ok = true;
                  else
                        ok = false;
            }
      }
      return(ok);
}
```

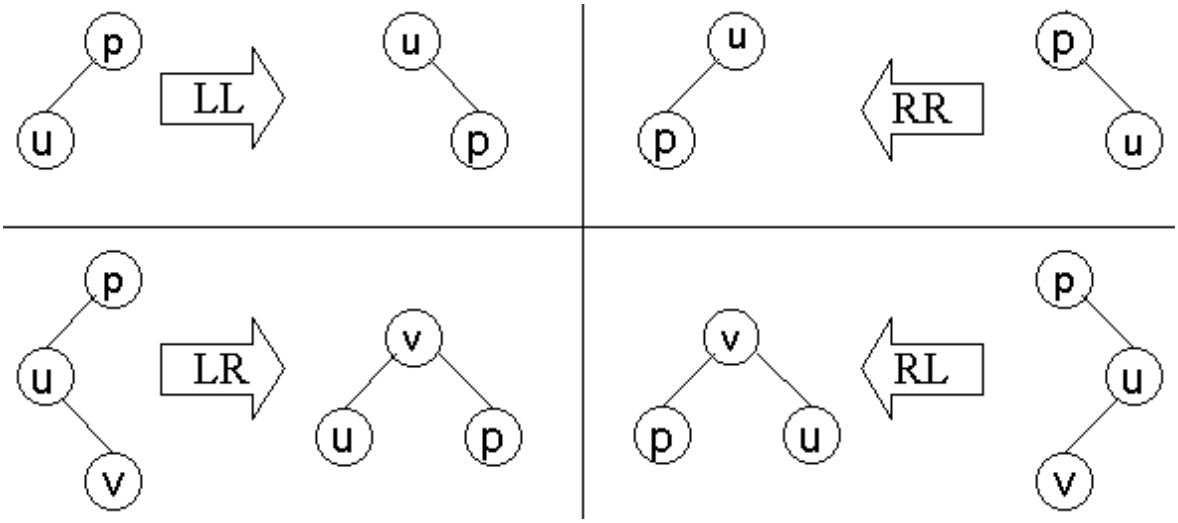
Operações de Rotação AVL

Conhecendo-se a posição x em que um nó foi inserido, é preciso verificar os ancestrais de x em busca de um possível nó p que tenha sido desregulado após esta operação. O nó p que deve ser encontrado é o mais próximo possível da folha x, ou seja, a pesquisa é feita da folha em direção à raiz.

Se houver algum nó p desregulado no caminho entre x e a raiz, este deve sofrer uma operação de rotação para que seu equilíbrio seja restaurado (e para que a árvore volte a ser AVL). A operação de rotação gira em torno de p, e somente esta subárvore é afetada.

A posição do nó x que causou o desequilíbrio em relação ao nó desregulado p determina a operação necessária. A partir de p, é preciso verificar os dois níveis seguintes da estrutura na direção em que se encontra x. Dependendo da combinação esquerda (L, de “left”) ou direita (R, de “right”) que leva de p até x (ou a um de seus ancestrais), temos uma de quatro operações possíveis, divididas em duas rotações simples (LL ou RR) e duas duplas (LR ou RL).

Nas rotações simples (LL e RR), existe um nó u que é filho de p (na direção de x). Este u passa a ser a nova raiz da subárvore afetada. Nas rotações duplas (LR e RL), o nó u possui ainda um filho esquerdo ou direito denominado v, também localizado em direção ao nó x, que passa a ser a nova raiz da subárvore em questão (ou seja, substituindo p). A figura a seguir ilustra a modificação da estrutura após cada uma das rotações.




```
// Rotações à direita (LL e LR) devolvendo a nova raiz p da subárvore
NO* rotacaoL(NO* p)
{
      NO* u;
      NO* v;
      u = p->esq;
      if(u->bal == -1)  {
            // LL
            p->esq = u->dir;
            u->dir = p;
            p->bal = 0;
            p = u;
      }
      else {
            // LR
            v = u->dir;
            u->dir = v->esq;
            v->esq = u;
            p->esq = v->dir;
            v->dir = p;
            if(v->bal == -1) p->bal =  1;
            else p->bal = 0;
            if(v->bal ==  1) u->bal = -1;
            else u->bal = 0;
            p = v;
      }
      p->bal = 0;       // balanço final da raiz (p) da subarvore
      return (p);
}
```

```
// Inserção AVL: p é inicializado com raiz e *ajustar com false.
// O retorno da função de mais alto nível é a raiz da árvore, possivelmente
// alterada caso tenha sofrido rotação ou seja a primeira inserção da estrutura.
// Assim, a chamada principal deve ser raiz = inserirAVL(raiz,ch,&ajustar);

NO* inserirAVL(NO* p, int ch, bool *ajustar) {
     if(!p) {
            p = (NO *) malloc(sizeof(NO));
            p->esq   = NULL;
            p->dir   = NULL;
            p->chave = ch;
            p->bal   = 0;
            *ajustar = true;
     }
     else   {
            if(ch < p->chave) {
                  p->esq = inserirAVL(p->esq, ch, ajustar);
                  if(*ajustar)
                        switch (p->bal) {
                              case 1 : p->bal = 0;
                                       *ajustar = false;
                                       break;
                              case 0 : p->bal = -1;
                                       break; // continua verificando
                              case -1: p = rotacaoL(p);
                                       *ajustar = false;
                                       break;
                        }
             }
             else {
                  p->dir = inserirAVL(p->dir, ch, ajustar);
                  if(*ajustar)
                          switch (p->bal) {
                              case -1: p->bal = 0;
                                       *ajustar = false;
                                       break;
                              case 0 : p->bal = 1;
                                       break; // continua verificando
                              case 1 : p = rotacaoR(p);
                                       *ajustar = false;
                                       break;
                          }
             }
     }
   return (p);
}
```

Observação: a variável ***ajustar*** é utilizada para indicar se é preciso verificar o balanceamento (e eventuamente comandar uma rotação) nos ancestrais do nó inserido. Esta variável se torna *true* no momento que o novo nó é criado, e se torna novamente *false* (encerrando portanto a verificação dos ancestrais) se o balanceamento não foi afetado pela inserção, ou se o equilíbrio foi restaurado através de uma rotação.