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 {
        TIPOCHAVE 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(TIPOCHAVE 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
TIPOCHAVE sairFdinam(Fdinam *f)
{
    NO* aux;
    TIPOCHAVE 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  {
        TIPOCHAVE 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(TIPOCHAVE 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
TIPOCHAVE 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, mas a dinâmica é mais comum, tirando proveito do encadeamento duplo para permitir acesso a ambas extremidades em tempo O(1).

```
typedef struct estrutura {
        TIPOCHAVE 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(TIPOCHAVE 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
TIPOCHAVE 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 {
        TIPOCHAVE 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(TIPOCHAVE 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
TIPOCHAVE pop(Pdinam *p) {
    NO* aux;
    TIPOCHAVE 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 {
        TIPOCHAVE 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(TIPOCHAVE 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
TIPOCHAVE pop(Pestat *p) {
     if(p->topo < 0) return (-1);
     TIPOCHAVE 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;
        TIPOCHAVE 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(TIPOCHAVE 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
TIPOCHAVE popK(PILHADUPLA *p, int k) {
      TIPOCHAVE 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];
        TIPOCHAVE 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, a qual não precisa 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(TIPOCHAVE 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.

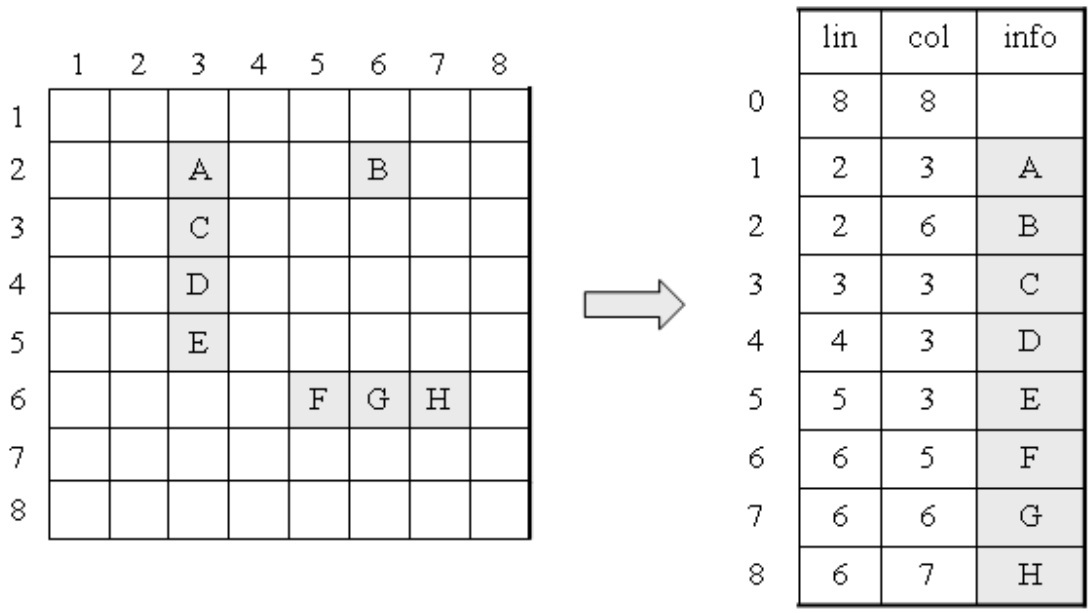
```
// Retirar um item da pilha k, ou -1
TIPOCHAVE popK(PILHAS *p, int k) {
      TIPOCHAVE resp = -1;
      if( (p->topo[k] > p->base[k]) )  {
            resp = p->A[p->topo[k]];
            p->topo[k]--;
      }
      return(resp);
}
```

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 (na verdade, 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.



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

	lin	col	info
0	8	8	
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;
} MATRIZ;
```

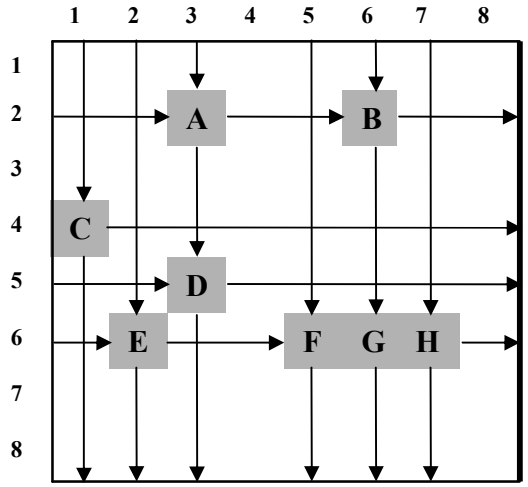
A economia de espaço desta representação é bastante significativa. Para uma matriz de MAXLIN x MAXCOL elementos, dos quais n são não-nulos, seu uso será vantajoso (do ponto de vista da complexidade de espaço) sempre que:

```
(MAXCOL*MAXLIN*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.

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

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(MATRIZ *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(MATRIZ 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(MATRIZ 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 tag cuja manutenção é responsabilidade do programador. Dependendo do valor de tag, armazenamos em um campo de tipo variável (um *union* em C) o dado correspondente. Para evitar a criação de códigos especiais para o tag, usamos a seguir uma enumeração dos dois valores possíveis (elemLista, inicioLista) para variáveis do tipo IDENT. Note no entanto que o uso de um tipo enumerado não é obrigatório e de fato não tem relação com o uso de *union*.

```
typedef enum{elemLista, inicioLista} IDENT;

typedef struct estrutura {
        IDENT tag;
        union {
              TIPOCHAVE 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->tag == elemLista)
               chaves++;
            else
               chaves = chaves + contarChaves(p->sublista);
            p = p->prox;
    }
    return(chaves);
}
```

```
// Quantidade de nos na lista
int contarNos(NO* p) {
      int nos = 0;
      while (p)   {
            nos++;
            if( p->tag == inicioLista)
               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->tag == elemLista) 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;
      IDENT tipo;
      novo = NULL;
      if (p) {
            tipo = p->tag;
            if( tipo == inicioLista)
                 abaixo = copiarListaGen(p->sublista);
            dir = copiarListaGen(p->prox);
            novo = (NO *) malloc(sizeof(NO));
            novo->tag = tipo;
            novo->prox = dir;
            if( tipo == elemLista)
                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->tag == b->tag) {
                  if( a->tag == elemLista)
                        resp = (a->chave == b->chave);
                  else
                        resp = listasIguais(a->sublista, b->sublista);
                  if(resp)
                        resp = listasIguais(a->prox, b->prox);
            }
      }
      return(resp);
}
```