Algoritmos e Complexidade

Estruturas de Dados

José Bernardo Barros
Departamento de Informática
Universidade do Minho

Conteúdo

1 Conjuntos e Multi-conjuntos **1**

2 Sequências **4**

3 Buffers **19**
3.1 Stacks 19
3.1.1 Usando arrays estáticos 21
3.1.2 Usando listas ligadas 23
3.1.3 Usando arrays dinâmicos 23
3.2 Queues 26
3.2.1 Usando arrays estáticos 27
3.2.2 Usando listas ligadas 30
3.3 Filas com prioridades 31

4 Dicionários **37**
4.1 Tabelas de Hash 37
4.1.1 Closed Addressing 38
4.1.2 Open Addressing 41
4.2 Árvores de procura balanceadas 49

A Exercícios de testes **60**

1 Conjuntos e Multi-conjuntos

As operações fundamentais que se pretendem implementar em conjuntos são a procura e inserção de um elemento no conjunto. Para além destas é normal disponibilizar ainda operações de união, intersecção e diferença de conjuntos.

```
void initSet (SetInt);
int  searchSet (SetInt, int);
int  addSet (SetInt, int);
int  emptySet (SetInt);
```

```
void unionSet (SetInt, SetInt, SetInt);
void intersectSet (SetInt, SetInt, SetInt);
void differenceSet (SetInt, SetInt, SetInt);
```

Um multi-conjunto é uma generalização do conceito de conjunto em que se tem em consideração a multiplicidade com que um elemento nele ocorre.
As operações a disponibilizar para tal tipo de dados são a generalização das correspondentes oerações sobre conjuntos.

```
void initMSet (MSetInt);
int  searchMSet (MSetInt, int);
int  addMSet (MSetInt, int);
int  emptyMSet (MSetInt);
void unionMSet (MSetInt, MSetInt, SetInt);
void intersectMSet (MSetInt, MSetInt, MSetInt);
void differenceMSet (MSetInt, MSetInt, MSetInt);
```

No caso de o universo ser um segmento inicial do conjunto dos números naturais a forma mais eficiente de implementar conjuntos (ou multi-conjuntos) é usar um array em que a posição `i` indica se o elemento `i` pertence ao conjunto ou, no caso dos multi-conjuntos, a multiplicidade com que ele ocorre.

Exemplo 1 O conjunto $\{2, 4, 10\}$ seria implementado num array (com pelo menos 11 posições) em que todas as componentes serão 0 excepto as posições 2, 4 e 10 que terão o valor 1.

0	1	2	3	4	5	6	7	8	9	10	11	...
0	0	1	0	1	0	0	0	0	0	1	0	0

Exemplo 2 O multi-conjunto $\{2, 2, 4, 4, 4, 10\}$ seria implementado num array (com pelo menos 11 posições) em que todas as componentes serão 0 excepto as posições 2, 4 e 10 que terão os valor 2, 3 e 1 respectivamente.

0	1	2	3	4	5	6	7	8	9	10	11	...
0	0	2	0	3	0	0	0	0	0	1	0	0

As definições seguintes definem estas representações de conjuntos e multi-conjuntos.

```
#define MAXS 100
typedef char SetInt [MAXS];
```

```
#define MAXMS 100
typedef int MSetInt [MAXS];
```

As operações de inicialização deverão assegurar que o conjunto passa a representar o conjunto vazio. Por isso deverão inicializar todas as posições dos arrays a zero.

```
int emptySet (SetInt s){
  int i;
  int r;
  for (i=0;
       i<MAXS;
       s[i++]=0)
    ;
}
```

```
void initMSet (MSetInt s){
  int i;
  for (i=0;
       i<MAXMS;
       s[i++]=0)
    ;
}
```

Da mesma forma, para verificar que um conjunto é vazio temos que percorrer todo o array que o representa.

```
void emptySet (SetInt s){
  int i; int r=1;
  for (i=0;
       i<MAXS && r == 1;
       i++)
    if (s[i]!=0) r=0;
  return r;
}
```

```
void emptyMSet (MSetInt s){
  int i; int r=1;
  for (i=0;
       i<MAXMS && r == 1;
       i++)
    if (s[i]!=0) r=0;
  return r;
}
```

O *custo* computacional destas duas operações de inicialização e teste é compensado pela eficiência das operações de adição de um novo elemento e de procura de um elemento.

```
int addSet (SetInt s,
            int x){
  s[x] = 1;
  return 0;
}
```

```
int addMSet (MSetInt s,
             int x){
  s[x] += 1;
  return 0;
}
```

```
int searchSet (SetInt s,
               int x){
  return (s[x]);
}
```

```
int searchMSet (MSetInt s,
                int x){
  return (s[x]);
}
```

Para definir as restantes operações sobre conjuntos teremos que voltar a percorrer a totalidades dos arrays que os representam.
Note-se que tal como acontece com os conjuntos, a união de dois multi-conjuntos deve corresponder ao **menor** multiconjunto que contém os dois.

```
void unionSet (SetInt s1,
               SetInt s2,
               SetInt r){
  int i;
  for (i=0; i<MAXS; i++)
     r[i]=s1[i] || s2[i];
}
```

```
void unionMSet (MSetInt s1,
                MSetInt s2,
                MSetInt r){
  int i;
  for (i=0; i<MAXS; i++)
     if (s1[i] > s2[i])
          r1[i] = s1[i];
     else r1[i] = s2[i];
}
```

De igual forma, a intersecção de dois multi-conjuntos deve corresponder ao **maior** multiconjunto contido nos dois.

```
void intersectSet (SetInt s1,
                   SetInt s2,
                   SetInt r){
  int i;
  for (i=0; i<MAXS; i++)
     r[i]=s1[i] && s2[i];
}
```

```
void intersectMSet (MSetInt s1,
                    MSetInt s2,
                    MSetInt r){
  int i;
  for (i=0; i<MAXS; i++)
     if (s1[i] > s2[i])
          r1[i] = s2[i];
     else r1[i] = s1[i];
}
```

```
void differenceSet (SetInt s1,
                    SetInt s2,
                    SetInt r){
  int i;
  for (i=0; i<MAXS; i++)
     r[i]=s1[i] && !(s2[i]);
}
```

```
void differenceMSet (MSetInt s1,
                     MSetInt s2,
                     MSetInt r){
  int i;
  for (i=0; i<MAXS; i++)
     if (s1[i] > s2[i])
          r[i]=s1[i] - s2[i];
     else r[i]=0;
}
```

Estas implementações só são possíveis quando o universo é um segmento inicial do conjunto dos números naturais. Além disso só é praticável se esse universo não fôr muito grande. Nos casos em que tal não acontece é costume guardar apenas a informação acerca dos elementos que pertencem ao conjunto. Veremos mais adiante (secção 4) como tal pode ser implementado de uma forma eficiente.

2 Sequências

Talvez a forma mais directa de idealizar um tipo de dados capaz de armazenar um número variável de itens seja como uma sequência. Este tipo de dados, primitivo na generalidade das linguagens funcionais, pode ser implementado numa linguagem imperativa como C usando a capacidade de alocar e libertar memória durante a execução dos programas/funções.
Relembremos em primeiro lugar as funções primitivas de gestão de memória disponíveis na linguagem C (cujo tipo se encontra definido em `stdlib.h`).

- `void *malloc(size_t size);` reserva um bloco contíguo de memória com `size` bytes. A função retorna o valor `NULL` sse não for possível alocar esse bloco de memória.
- `void free(void *ptr);` liberta a memória (previamente alocada usando uma invocação de `malloc`) que se inicia no endereço `ptr`.

Para além disso convém referir a existência da constante `NULL` que representa um *endereço de memória inválido.*

Vejamos então como definir um tipo capaz de representar listas de inteiros.

```
typedef struct lista *LInt;

struct lista {
  int valor;
  LInt prox;
};
```

Uma definição alternativa e equivalente seria

```
typedef struct lista {
  int valor;
  struct lista *prox;
} *LInt;
```

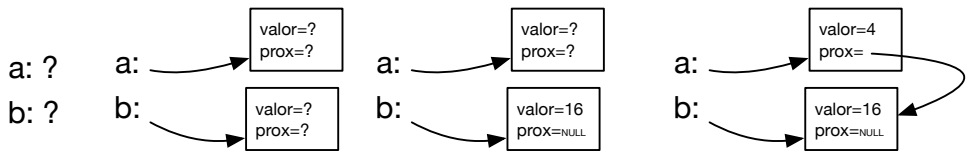
Para melhor compreendermos como com esta definição é possível representar qualquer lista de inteiros podemos fazer as seguintes observações:

- A lista vazia será representada pela constante `NULL`.
- Uma lista não vazia será representada pelo endereço (de uma `struct lista`) onde estão guardados: o valor do primeiro elemento da lista (campo `valor`) e a lista dos restantes elementos (campo `prox`).

Exemplo 3 Consideremos o seguinte excerto de código.

```
LInt a,b;
a=malloc (sizeof (struct lista));
b=malloc (sizeof (struct lista));
b->valor = 16;
b->prox=NULL;
a->valor = 4;
a->prox = b;
```

A evolução do estado pode ser visualizada no seguinte esquema.



Exercício 1 Apresente a evolução do estado quando alteramos o código do exemplo anterior para o seguinte.

```
LInt a,b;
a=b=malloc (sizeof (struct lista));
b->valor = 16;
b->prox=NULL;
a->valor = 4;
a->prox = b;
```

O exemplo anterior põe em evidência a forma como acrescentar um elemento no início de uma lista.

```
LInt cons (int x, LInt l) {
  LInt new = malloc (sizeof (struct lista));
  if (new != NULL) {
    new->valor=x;
    new->prox=l;
  }
  return new;
}
```

Exemplo 3 (continuação) Usando a função `cons` o excerto de código apresentado pode ser reescrito como.

```
LInt a,b;
b=cons(16,NULL);
a=cons(4,b);
```

A função seguinte calcula o comprimento de uma lista. Para isso vai *seguindo* os endereços dos vários elementos da lista.

```
int length (LInt l){
  int r=0;
  while (l!=NULL) {
    r++; l=l->prox;
  }
  return r;
}
```

Tal como vimos atrás, a operação de acrescentar um elemento no início de uma lista é bastante simples e executa em tempo constante, i.e., o número de operações a efectuar não depende do número de elementos da lista.
Já a operação de acrescentar um elemento no final da lista vai precisar de percorrer toda a lista, tendo por isso um comportamento linear no número de elementos da lista argumento.
Vejamos uma primeira tentativa (errada) de resolver este problema.

```
LInt snoc (int x, LInt l) {
  LInt pt = l;
  while (pt != NULL) pt = pt->prox;
  pt = malloc (sizeof (struct lista));
  pt->valor=x;
  pt->prox=NULL;
  return l;
}
```

Esta definição, apesar de percorrer a lista toda acaba por não fazer qualquer alteração à lista recebida. Após terminar o ciclo a variável `pt` tem o valor `NULL` e as restantes

operações apenas modificam o valor dessa variável, sem ligar qualquer nodo adicional à lista.
Uma forma de solucionar este problema passa por evitar que o ciclo execute a última iteração. Ou pelo menos que guarde o último valor que a variável `pt` tenha assumido antes de atingir `NULL`.
Vejamos então como modificar a definição acima de forma a incorporar esta observação.

```
LInt snoc (int x, LInt l) {
  LInt pt = l, ant;
  while (pt != NULL) {
    ant = pt; pt = pt->prox;
  }
  pt = malloc (sizeof (struct lista));
  pt->valor=x;
  pt->prox=NULL;
  ant->prox=pt;
  return l;
}
```

A instrução `ant = pt;` imediatamente antes de actualizar o valor de `pt` garante-nos que no final do ciclo (`pt==NULL`)) a variável `ant` tem o endereço da última célula da lista. Isto se a lista original fosse não vazia, caso contrário o valor da variável `ant` é indefinido. Podemos testar este caso particular antes de fazer a atribuição final.

```
LInt snoc (int x, LInt l) {
  LInt pt = l, ant;
  while (pt != NULL) {
    ant = pt; pt = pt->prox;
  }
  pt = malloc (sizeof (struct lista));
  pt->valor=x;
  pt->prox=NULL;
  if (l==NULL) l=pt;
  else ant->prox=pt;
  return l;
}
```

Uma alternativa a esta definição passa por começar por isolar o caso particular de se tratar de uma lista vazia. No caso em que a lista não é vazia, podemos percorrer a lista até encontrarmos o último elemento (aquele cujo campo `prox` é não nulo) em vez de percorrermos a lista até ao fim.

```
LInt snoc (int x, LInt l) {
  LInt new, pt;
  new=malloc (sizeof (struct lista));
  new->valor=x;
  new->prox=NULL;

  if (l==NULL) l=new;
```

```
  else {
    pt = l;
    while (pt->prox != NULL)
        pt=pt->prox;
    pt->prox=new;
  }
  return l;
}
```

Em ambas as alternativas apresentadas o caso da lista vazia tem que ser tratado como um caso particular. Isto é de alguma forma natural: apenas no caso da lista ser vazia é que o início da lista (e daí o valor de retorno da função) é diferente do valor recebido como argumento.
No entanto esat singularidade pode ser evitada se abordarmos o problema de uma forma ligeiramente diferente.
Em ambos os casos, i.e., quer a lista seja vazia ou não, aquilo que esta função deve fazer é colocar um novo nodo a ser apontado por algo que até aí tinha o valor `NULL`:

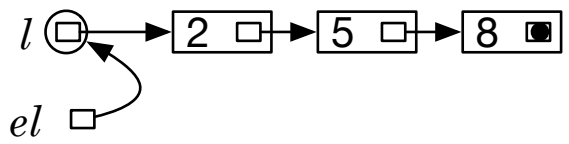
- no caso da lista `l` ser vazia, é `l` que será alterado
- no caso da lista não ser vazia, o que será alterado será o campo `prox` do último nodo da lista.

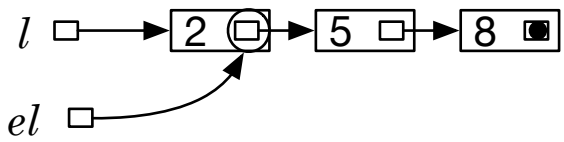
Para tratarmos ambos os casos da mesma forma, vamos usar uma variável do tipo *endereço de lista* onde vamos calcular o endereço de memória que será actualizado.

```
LInt snoc (int x, LInt l) {
  LInt *el = &l;

  while (*el != NULL)
        el=&((*el)->prox);

  *el = malloc (sizeof (struct lista));
  (*el)->valor=x;
  (*el)->prox=NULL;

  return l;
}
```

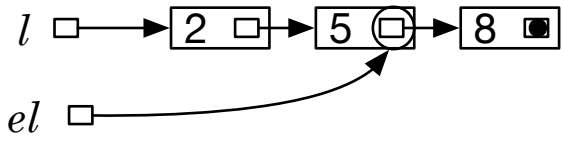
Exemplo 4 Vejamos o que acontece quando esta função é invocada para acrescentar o número 9 a uma lista com 3 elementos (2, 5 e 8).

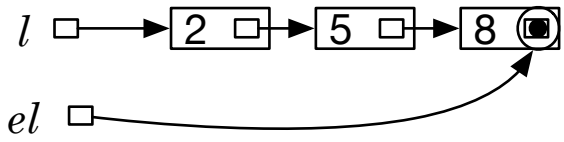
- A variável `el` começa por ser inicializada com o endereço de `l`.



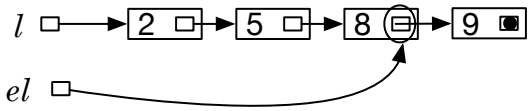


- De seguida vai sendo actualizada até que está a endereçar um valor `NULL` (ciclo `while`).





- Finalmente essa posição de memória que continha `NULL` vai ser actualizada com o endereço de uma lista apenas com o elemento a acrescentar.



Exercício 2 Apresente definições das funções

- `int push (LInt *l, int x)`
- `int append (LInt *l, int x)`

alternativas às funções `cons` e `snoc` respectivamente, que alteram o valor da lista passada como argumento (por referência) em vez de retornar a lista calculada.

Exercício 3 Defina a função `LInt concatL (LInt a, LInt b)` que acrescenta a segunda lista (`b`) à primeira (`a`) retornando a lista resultante.
Apresente ainda uma definição alternativa mas em que, em vez de retornar a lista resultante, recebe a primeira das listas por referência, i.e., com o tipo `void appendL (LInt *a, LInt b)`.

Vejamos agora como definir uma função para inverter a ordem dos elementos de uma lista. Uma possível estratégia consiste em, para listas não vazias, começar por inverter a ordem dos elementos da cauda da lista seguido de colocar o primeiro elemento da lista original no final.

```
LInt reverseL (LInt l) {
  LInt r, pt;

  if (l==NULL || l->prox==NULL) r=l;
  else {
    r = pt = reverseL (l->prox);
```

```
    while (pt->prox != NULL)
        pt=pt->prox;
    pt->prox=l;
    l->prox=NULL;
  }
}
```

Exercício 4 Mostre que a função `reverseL` acima tem uma complexidade quadrática em função do número de elementos da lista argumento.
Para isso comece por apresentar uma relação de recorrência que traduza essa complexidade e resolva essa recorrência.

Uma forma alternativa e mais eficiente de definir esta função deverá evitar ter que percorrer a lista que está a ser construída.
Para isso podemos percorrer a lista argumento e, para cada elemento (e por essa ordem) acrescentá-la no início de uma lista inicialmente vazia.

```
LInt reverseL (LInt l) {
  LInt r, tmp;
  r=NULL;

  while (l!=NULL) {
    tmp=l; l=l->prox;
    tmp->prox=r; r=tmp;
  }
  return r;
}
```

Exercício 5 Apresente (informalmente) um invariante do ciclo acima que lhe permita provar a correcção da função, face à seguinte especificação:

Pré-condição: $l == l_0$

pós-condição: $r ==$ rev (l_0)

Vimos como acrescentar um elemento a uma lista quer no início (`cons` e `push`) quer no fim (`snoc` e `append`). A inserção noutro qualquer ponto da lista é muito semelhante à inserção no final da lista.
Vejamos por exemplo como fazer a inserção ordenada, i.e., como inserir um elemento numa lista ordenada (mantendo-a ordenada).
Tal como acontecia atrás (na inserção no fim) vamos ter que isolar o caso particular em que essa inserção se faz no início da lista: quer porque a lista é vazia quer porque o elemento a inserir é menor do que o primeiro elemento da lista.
No caso de tal não acontecer devemos começar por descobrir onde o novo elemento será inserido.

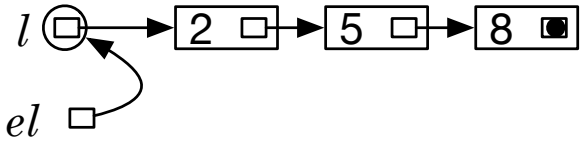
```
LInt insere (LInt l, int x){
  LInt new, pt, ant;
  new = malloc (sizeof (struct lista));
  new->valor=x;

  if (l==NULL || l->valor > x) {
      // insercao no inicio
      new->prox=l;
      l = new;
  } else {
      // posicionar na lista
      ant=l; pt = l->prox;
      while (pt != NULL && pt->valor < x){
          ant=pt; pt=pt->prox;
      }
      // ligar o novo nodo
      new->prox=pt;
      ant->prox=new;
  }
  return l;
}
```

Uma forma alternativa de definir esta função sem particularizar o caso de inserção no início da lista consiste em usar a mesma estratégia que usámos na função `snoc`: usar um endereço de uma lista para determinar onde se colocará o endereço da nova célula.

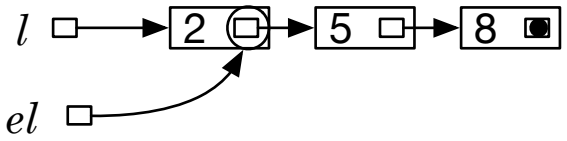
```
LInt insere (LInt l, int x){
  LInt new, *pt;

  new = malloc (sizeof (struct lista));
  new->valor=x;

  // posicionar na lista
  pt = &l;
  while (*pt != NULL && (*pt)->valor < x)
      pt=&((*pt)->prox);

  // ligar o novo nodo
  new->prox=*pt;
  *pt=new;

  return l;
}
```

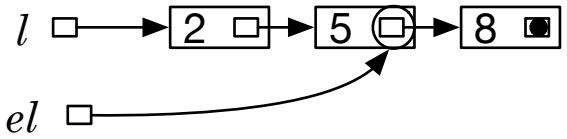
Exemplo 5 Vejamos o que acontece quando esta função é invocada para acrescentar o número 7 a uma lista com 3 elementos (2, 5 e 8).

- A variável `el` começa por ser inicializada com o endereço de `l`.

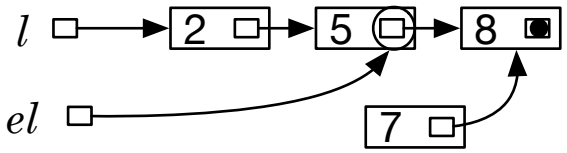


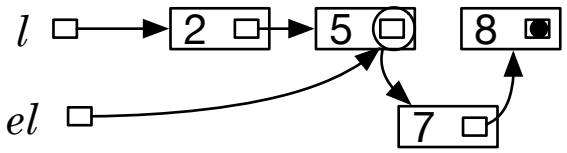
- De seguida ela vai sendo actualizada até que está a endereçar um valor que corresponde a uma lista que começa com um valor maior do que `x` (7) (ciclo `while`).





- Finalmente a nova célula é ligada à lista.





Vejamos agora mais um exemplo em que esta técnica de usar um *duplo apontador* (uma variável do tipo `*LInt` é um endereço do endereço onde se encontra o primeiro elemento de uma lista) elimina a necessidade de se isolar o caso particular do início da lista.
O problema em questão é o de produzir uma cópia de uma lista. Para isso vamos percorre a lista original e, para cada elemento acrescentar um novo nodo à lista que irá ser retornada.
De forma a não termos que percorrer esta sucessivamante guardaremos numa variável `pt` o endereço do último nodo que foi acrescentado. Daí que seja necessário inicializar essa lista com uma cópia do primeiro elemento da lista original e só depois copiar a restante lista.

```
LInt cloneL (LInt l) {
  LInt r, pt;

  if (l == NULL) r = NULL;
  else {
    r = pt = malloc (sizeof (struct lista));
```

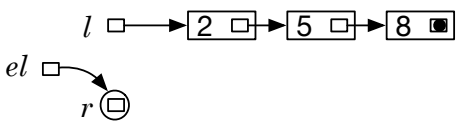
```
    r->valor = l->valor;
    l=l->prox;
    while (l!=NULL) {
      pt->prox = malloc (sizeof (struct lista));
      pt = pt->prox;
      pt->valor=l->valor;
      l=l->prox;
    }
    pt->prox=NULL;
  }
  return r;
}
```

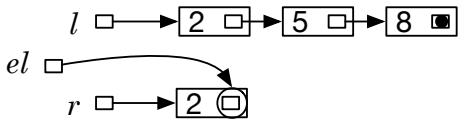
Usando um duplo apontador não temos que isolar a cópia do primeiro elemento da lista original da cópia dos restantes.

```
LInt cloneL (LInt l) {
  LInt r, *el;

  for (el = &r; l!=NULL; l=l->prox){
      *el = malloc (sizeof(struct lista));
      (*el)->valor=l->valor;
      el=&((*el)->prox);
  }
  *el=NULL;
  return r;
}
```

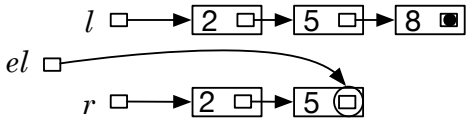
Exemplo 6 Vejamos o que acontece quando esta função é invocada para duplicar uma lista com 3 elementos (2, 5 e 8).

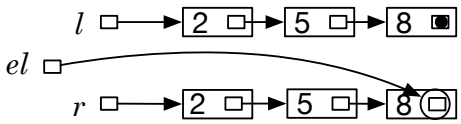
- A variável `el` é inicializada com o endereço de `r` – a variável cujo valor será retornado no final; esta é a próxima variável a ser preenchida.



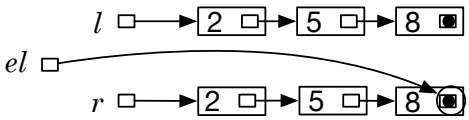


- Por cada elemento da lista é criado um novo nodo com o valor correspondente e actualizado `el` com o endereço que deve ser actualizado de seguida.





- A função termina colocando `NULL` no último endereço a ser actualizado.



Um outro exemplo onde o uso de um duplo apontador torna o código bastante mais compacto é na definição de uma função que faz o *merge* de duas listas ordenadas numa única lista ordenada.
A forma normal de executar esta operação consiste em percorrer (simultaneamente) as duas listas passando para (o fim) da lista resultado o menor entre os primeiros elementos das listas argumento.
Este processo termina quando uma das listas acaba, e nesse ponto a outra lista é acrescentada no final.
Mais uma vez, uma vez que a lista resultado é cosntruída adicionando nodos no final, convem-nos manter o endereço do último nodo acrescentado (para evitar ter que percorrer sucessivamente essa lista).
Tal como acontecia no exemplo anterior, a construção do primeiro elemento da lista resultado vai ter que ser feita à parte (i.e., fora do ciclo) uma vez que será nessa altura que se inicializa o dito endereço do último nodo acrescentado.

```
Lint mergeL (LInt a, LInt b) {
    LInt r, pt;
    if (a==NULL) r=b;
    else if (b==NULL) r=0;
    else {
      if (a->valor < b->valor) {
         r=a; a=a->prox;
      } else {
         r=b; b=b->prox;
      }
      pt=r;
```

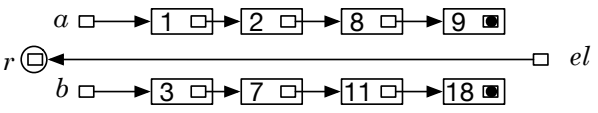
```
    while (a!=NULL && b!=NULL) {
        if (a->valor < b->valor){
            pt->prox=a; a=a->prox;
        } else {
            pt->prox=b; b=b->prox;
        }
        pt=pt->prox;
    }
    if (a==NULL) pt->prox=b;
    else pt->prox=a;
  }
  return r;
}
```

Vejamos agora como o uso de um duplo apontador (que irá conter o endereço que devemos actualizar a cada instante) elimina muitos dos casos particulares que tivemos que considerar acima.

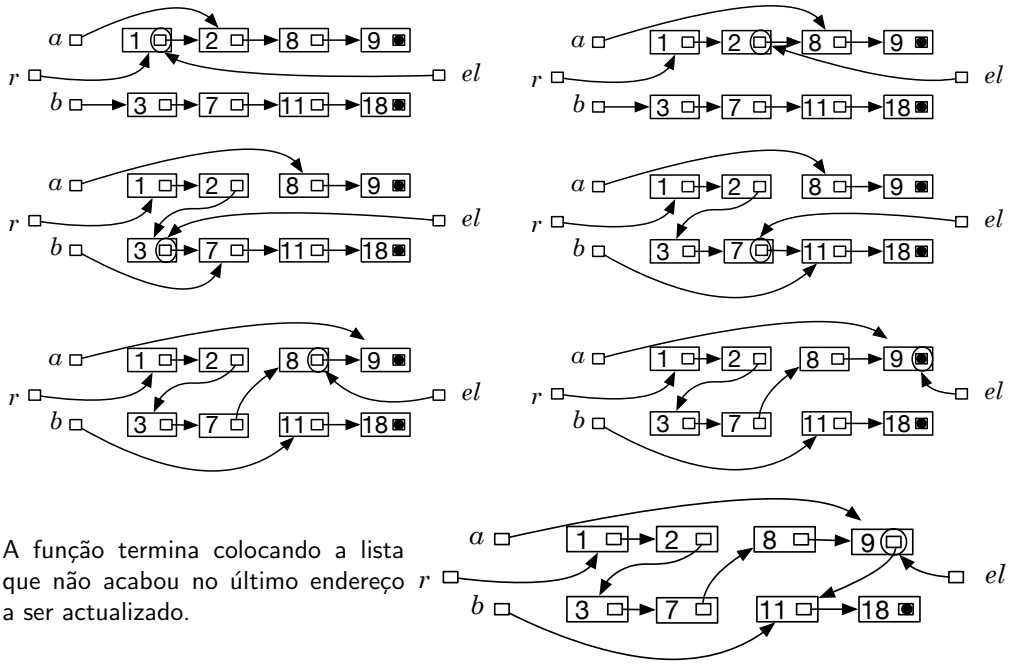
```
LInt mergeL (LInt a, LInt b) {
  LInt r, *el;
  el=&r;
  while (a!=NULL && b!=NULL) {
    if (a->valor < b->valor){
        *el=a; a=a->prox;
    } else {
        *el=b; b=b->prox;
    }
    el=&((*el)->prox);
  }
  if (a==NULL) *el=b;
  else *el=a;

  return r;
}
```

Exemplo 7 Vejamos o que acontece quando esta função é invocada para juntar uma lista a com os elementos 1, 2, 8 e 9 com uma lista b cujos elementos são 3, 7, 11 e 18.

- A variável `el` é inicializada com o endereço de `r` – a variável cujo valor será retornado no final; esta é a próxima variável a ser preenchida.



- Enquanto nenhuma das listas chega ao fim, é colocado em `*el` o endereço do menor dos valores, avançando a lista correspondente e actualizado `el` com o endereço que deve ser actualizado de seguida.



- A função termina colocando a lista que não acabou no último endereço a ser actualizado.

De seguida vamos apresentar a definiçao de uma função `LInt splitL (LInt l)` que, dada uma lista ligada `l` retorna a lista com os elementos de `l` que estão nas posições pares, deixando em `l` apenas os restantes (i.e., os que se encontram nas posições ímpares). Por exemplo, se a lista `a` contiver os elementos 10, 20, 30, 40 e 50, por esta ordem, após a invocação de `b=splitL (a)` as listas `a` e `b` devem ter os valores

- `a`: 10, 30 e 50
- `b`: 20 e 40

Mais uma vez, o uso de um duplo apontador permite-nos definir esta função de uma forma bastante compacta.

```
LInt splitL (LInt l){
  LInt r, *el;

  el = &r;

  while (l!=NULL && l->prox!=NULL) {
     *el = l->prox; el = &((*el)->prox);
     l->prox = l->prox->prox;
     l=l->prox;
  }
  *el = NULL;
  return r;
}
```

É de notar que esta função retorna `NULL` exactamente quando a lista argumento tem menos do que dois elementos. Esse facto vai-nos permitir apresentar uma versão muito simples da função `mergeSL` de ordenação de listas por *merge-sort*.

```
LInt mergeSL (LInt l) {
  LInt tmp;
  tmp = splitL (l);
  if (tmp!=NULL)
     l=mergeL (mergeSL(l), mergeSL(tmp));
  return l;
}
```

Uma outra forma de ordenar listas é usando o algoritmo de quick-sort. Aqui começa por separar a lista original em duas em que os elementos de uma são menores do que os elementos da outra. O processo evolui por ordenação destas duas componentes seguido da concatenação das listas resultantes.
Sopunhamos então que exste definida uma função `LInt split (LInt l)` que recebe uma lista **não vazia** e remove dessa lista todos os elementos que são menores do que o primeiro elemento da lista (retornando a lista dos elementos removidos).
A definição da função `qSortL` poderá ser então escrita usando essa função e a função `concatL` referida no exercício .

```
LInt qSortL (LInt l) {
  LInt tmp;
  if (l!=NULL) {
     tmp=split (l);
     l->prox=qSortL(l->prox);
     tmp=qSortL (tmp);
     l=concatL(tmp,l);
  }
  return l;
}
```

Exercício 6 Admitindo que as funções `split` e `concatL` executam em tempo linear (N) no comprimento da lista argumento, faça uma análise da complexidade da função `qSortL` definida acima.
Para isso identifique o melhor e pior caso da execução da função, e analise o comportamento da função em cada um desses casos.

Uma das desvantagens desta definição relativamente à definição da correspondente função em arrays é que aqui a junção das duas componentes ordenadas (`concatL` tem uma complexidade não constante.
Uma forma de minimizar essa desvantagem consiste em calcular também o endereço do último nodo da lista. Dessa forma conseguimos passar para constante o esforço de concatenar as listas.

```
LInt qSortL (LInt l) {
  LInt aux;
```

```
  return (qSortLAux (l, &aux));
}

LInt qSortLAux (LInt l, LInt *end){
   LInt e1, e2, tmp;

  if (l==NULL) *end = NULL;
  else {
     tmp=split (l);
     l->prox=qSortLAux(l->prox, &e1);
     tmp=qSortLAux (tmp, &e2);
     if (e1==NULL) *end=l; else *end=e1;
     if (e2!=NULL) {
        e2->prox=l; l= tmp;
     }
  }
  return l;
}
```

Exercício 7 Apresente uma definição da função `LInt split (Lint l)` usada acima e que recebe uma lista **não vazia** e remove dessa lista todos os elementos que são menores do que o primeiro elemento da lista (retornando a lista dos elementos removidos).

Exercício 8
Considere que se usa uma lista ligada para contar o número de palavras de um texto.
Esta lista ligada pode estar organizada de várias formas.

- por ordem alfabética
- por ordem de (primeira) ocorrência no texto.

```
typedef struct cref {
  char pal [50];
  int cont;
  struct cref *prox;
} *CRef;
```

Considere ainda a função `int ocorre (CRef *c, char p[])` que regista uma nova ocorrência da palavra `pal` na lista `*c`.
Apresente duas definições alternativas dessa função de acordo com as organizações referidas.
Note que no segundo caso, caso a palavra a registar não ocorra ela deverá ser inserida no final da lista.

Uma alternativa às definições referidas no exercício anterior consiste em, sempre que um nodo é inserido ou modificado, esse nodo passa a estar no início da lista (*move to front*).

```
int ocorre (CRef *c, char p[]) {
  CRef pt, ant;
  pt=*c; ant=NULL;
  while (pt!=NULL && strcmp (pt->pal, p)!=0) {
      ant=pt; pt=pt->prox;
  }
```

```
  if (pt==NULL) {
      pt = malloc (sizeof (struct cref));
      strcpy (pt->pal,p); pt->cont=1;
  } else {
      pt->cont++;
      if (ant==NULL) *c=pt;
      else ant->prox=pt;
    }
  pt->prox=*c; *c=pt;
  return (pt->cont);
}
```

Na definição acima podemos identificar 3 fases distintas:

1. (`while`) Começamos por percorrer a lista de forma a identificar o nodo onde se encontra a palavra a processar, ou concluir que tal nodo não existe.

2. (`if`) Dividimos a informação em duas componentes: a lista sem a palavra em causa e o nodo com essa palavra.

3. Construimos o resultado final usando as duas componentes obtidas na fase anterior.

3 Buffers

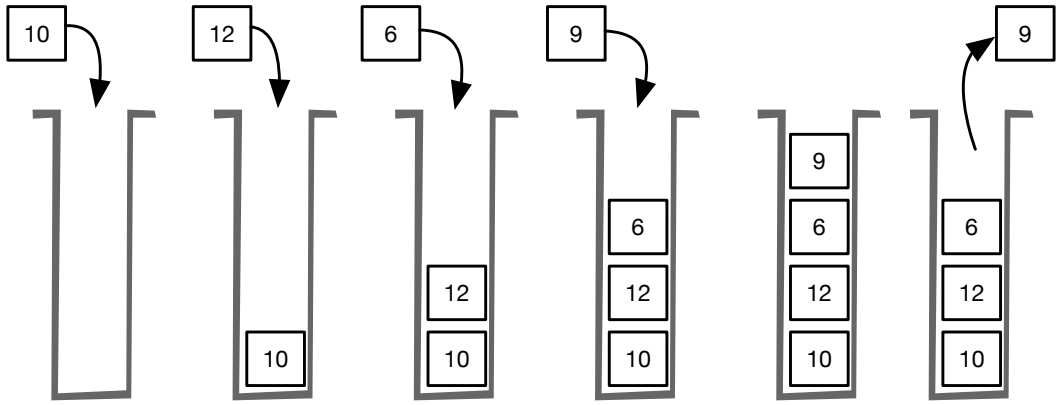
Um *buffer* é um tipo de dados usado para armazenar itens e em que as operações fundamentais são a adição e a remoção de um elemento.
Uma das principais diferenças relativamente a outros tipos abstractos de dados (tais como conjuntos os sequências) é a ausência de uma operação de procura ou de remoção de um item particular. Em vez disso existe uma operação de remoção do próximo elemento. A definição de qual é o próximo elemento a ser removido é aquilo que vai distinguir as várias instanciações de buffers.

3.1 Stacks

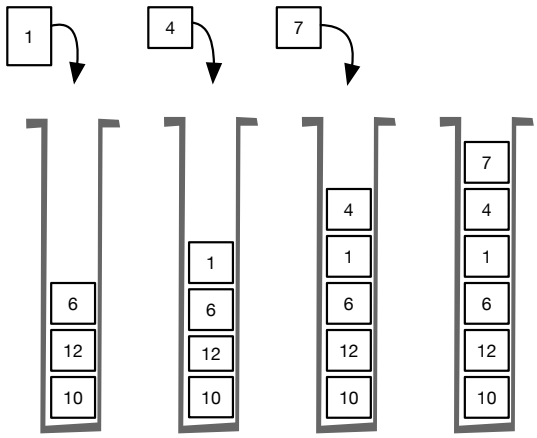
Uma stack é um buffer em que o próximo elemento a ser removido é o último elemento inserido (*Last In First Out*).
As operações de inserção e remoção de um elemento são normalmente referidas como **push** e **pop**.

Exemplo 8 Vejamos então o resultado de, numa stack de inteiros, inicialmente vazia, se efecturem algumas destas operações.

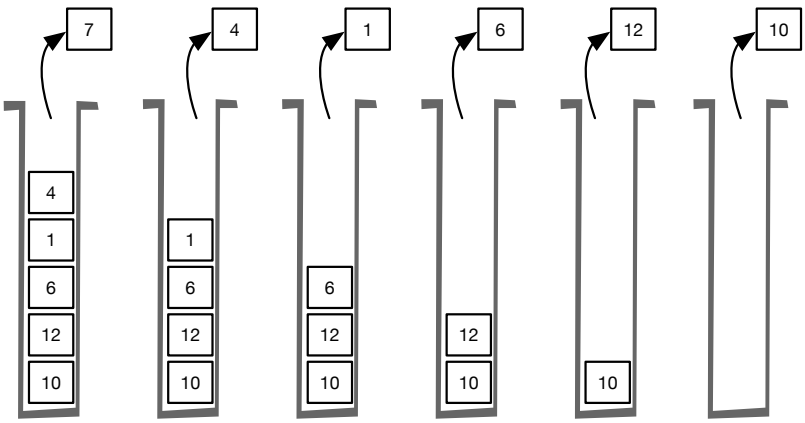
1. O resultado de acrescentar (`push`) os números 10, 12, 6 e 9 é



2. Ao removermos (pop) um elemento o elemento removido é o 9.
3. Após a inserção de 1 4 e 7 esta stack passa a ter 6 elementos.



4. Para a *esvaziar* será necessário fazer 6 remoções. A ordem pela qual os elementos serão removidos é 7, 4, 1, 6, 12 e 10.



3.1.1 Usando arrays estáticos

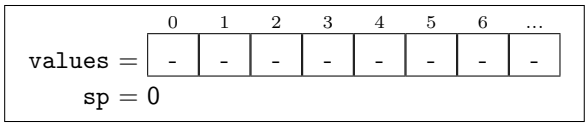
A primeira implementação de stacks que vamos apresentar usa um array (onde os elementos da stack são armazenados por ordem de entrada) e um inteiro que nos indica quantos elementos se encontram em cada momento na stack.

```
#define MAX 1000
typedef struct stack {
  int values[MAX];
  int sp;
} Stack;
```

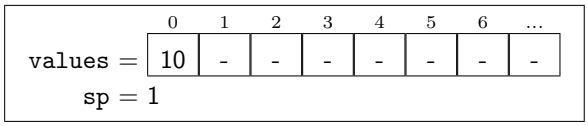
De forma a obter versões eficientes das diferentes operações é conveniente que os elementos sejam armazenados no array por ordem cronológica: a inserção de um elemento far-se-á depois dos que já lá estão. Dessa forma, o último elemento inserido estará sempre na posição `sp-1`.

Exemplo 9 No exemplo apresentado acima, a evolução da stack será:

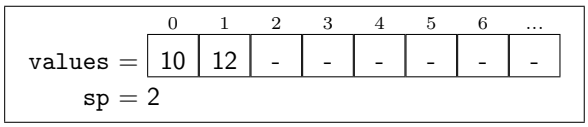
- stack vazia



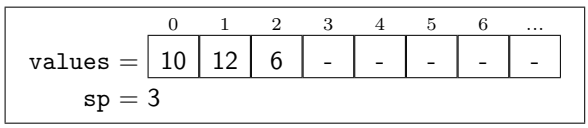
- `push(10)`



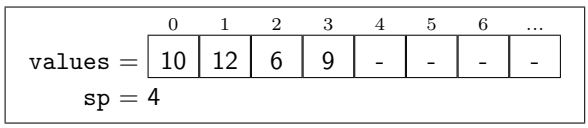
- `push(12)`



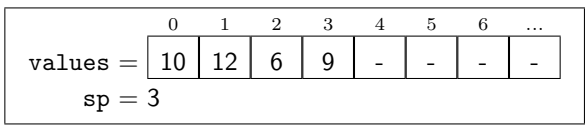
- `push(6)`



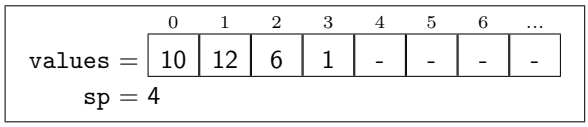
- `push(9)`



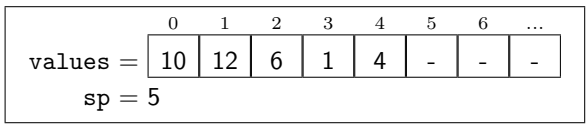
- `pop`



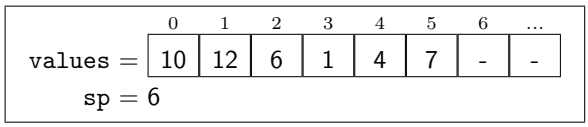
- `push(1)`



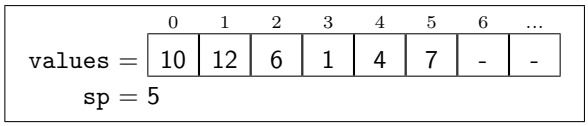
- `push(4)`



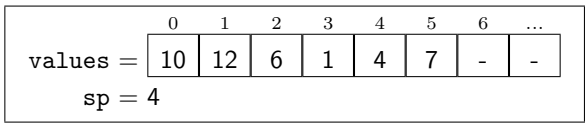
- `push(7)`



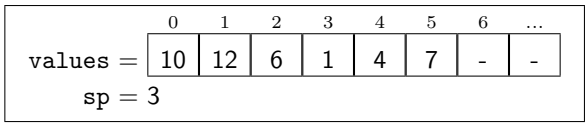
- `pop`



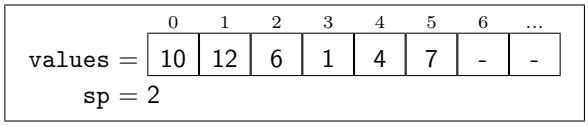
- `pop`



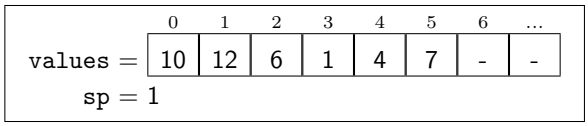
- `pop`



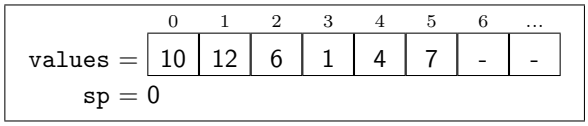
- `pop`



- `pop`



- `pop`



As funções de inicialização de uma stack e de teste se a stack está ou não vazia precisam apenas de manipular o campo `sp`.

```
void initStack (Stack *s){
  s->sp=0;
}
```

```
int isEmpty (Stack *s){
  return (s->sp==0);
}
```

As operações de `push` e `pop` retornam um código de sucesso uma vez que nem sempre são possíveis de efectuar. No caso da operação `push`, isso acontece quando a stack esgotou a sua capacidade. No caso do `pop` quando a stack está vazia.

```
int push (Stack *s, int x){
  int r=0;
  if (s->sp == MAX) r=1;
  else
    s->values[s->sp++] = x;
  return r;
}
```

```
int pop (Stack *s, int *x){
  int r=0;
  if (s->sp == 0) r=1;
  else
    *x = s->values[--s->sp];
  return r;
}
```

3.1.2 Usando listas ligadas

A grande desvantagem da implementação anterior é o facto de a capacidade da stack ser fixa.
A alternativa mais directa para ultrapassar esta limitação consiste em manter os elementos da stack numa lista ligada. As inserções e remoções de elementos fazem-se no início da lista e por isso continuam a ser muito eficientes.

```
typedef struct celula {
  int value;
  struct celula *prox;
} Celula, *Stack;
```

A stack vazia será representada por uma lista vazia.

```
void initStack (Stack *s){
  *s = NULL;
}
```

```
int isEmpty (Stack s){
  return (s==NULL);
}
```

As operações de `push` e `pop` retornam um código de sucesso uma vez que nem sempre são possíveis de efectuar. No caso da operação `push`, isso acontece quando já não há memória disponível. No caso do `pop` quando a stack está vazia.

```
int push (Stack *s, int x){
  int r=0;
  Celula *new;

  new=malloc(sizeof (Celula))
  if (new == NULL) r=1;
  else {
    new->value=x;
    new->prox=*s;
    *s=new;
  }
  return r;
}
```

```
int pop (Stack *s, int *x){
  int r=0;
  Celula *tmp;
  if (*s == NULL) r=1;
  else {
    *x = (*s)->value;
    tmp=*s;
    *s=(*s)->prox;
    free (tmp);
  }
  return r;
}
```

3.1.3 Usando arrays dinâmicos

Uma terceira alternativa que tenta combinar as vantagens das duas implementações apresentadas consiste em usar arrays dinâmicos.

```
typedef struct stack {
  int size;
  int *values;
  int sp;
} Stack;
```

A função de inicialização de uma stack tem que começar por alocar espaço para o array de valores.

```
void initStack (Stack *s, int n){
  s->size=n;
  s->values = malloc (n*sizeof(int));
  s->sp=0;
}

int isEmpty (Stack *s){
  return (s->sp==0);
}
```

As operações de push e pop poderão ter que re-dimensionar o array de valores. Na versão apresentada a seguir apenas a operação de push o faz (quando a stack está cheia, duplica o tamanho do array).

```
int push (Stack *s, int x){
  int r=0;
  if (s->sp == s->size) r = doubleArray (s);
  if (r == 0) s->values[s->sp++] = x;
  return r;
}

int pop (Stack *s, int *x){
  int r=0;
  if (s->sp == 0) r=1;
  else
    *x = s->values[--s->sp];
  return r;
}
```

A duplicação do array pode ser feita usando a função `realloc` que muda o espaço relativo a um endereço previamente alocado.

```
int doubleArray (Stack *s){
  int r=0;
  s->size *= 2;
  s->values = realloc (s->values, s->size);
  return r;
}
```

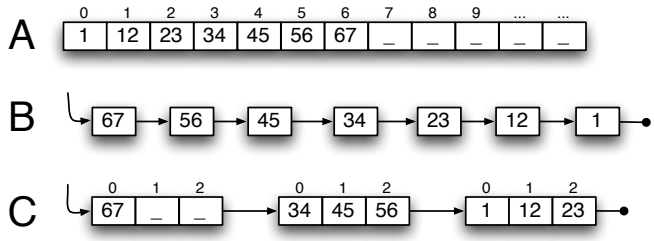
Uma outra alternativa, menos eficiente mas que evidencia os passos da solução anterior, consiste em usar a função `malloc` para reservar o novo espaço.

```
int doubleArray (Stack *s){
  int r=0; int i;
  int *newA;

  newA=malloc (2*s->size*sizeof(int));
```

```
  if (newA == NULL) r=1;
  else{
    for (i=0;i<s->size;i++)
       newA[i] = s->values[i];
    s->size*=2;
    free(s->values);
    s->values=newA;
  }
  return r;
}
```

Exercício 9 Considere que quando o número de elementos numa stack desce abaixo de 25% da sua capacidade esta deve ser redimensionada para metade do seu valor. Apresente as alterações necessárias à função pop acima.

Exercício 10 Considere um alternativa para implementar stacks de números inteiros que consiste em usar uma lista ligada de arrays (todos do mesmo tamanho – `MAXc`). A inserção de um novo elemento faz-se no final do primeiro array da lista se ainda não estiver cheio. Quando este se encontra cheio, é criado um novo array que é inserido no início da lista.



Na figura mostram-se as três alternativas que representam a stack (de inteiros) que resulta de, na stack vazia se acrescentarem os elementos 1, 12, 23, 34, 45, 56 e 67, por esta ordem. Note que nesta representação apenas temos que guardar o número de elementos do primeiro dos arrays da lista uma vez que todos os outros se encontram completos.
Para isso, a `struct StackC` guarda este número, juntamente com a lista dos arrays.
Defina as seguintes funções sobre esta implementação de stacks.

```
typedef struct chunk {
  int vs [MAXc];
  struct chunk *prox;
} *CList;
typedef struct stackC {
  CList valores;
  int sp;
} StackC;
```

1. `int push (StackC *s, int x)` que acrescenta um elemento `x` a `s`. Retorna `0` em caso de sucesso.
2. `int pop (StackC *s, int *x)` que remove o elemento do topo da stack, colocando-o em `*x`. Retorna `0` em caso de sucesso.
3. `int size(StackC s)` que calcula o comprimento (número de elementos) de `s`.

4. Usando as funções `push` e `pop` acima defina a função `void reverse (StackC *s)` que inverte a ordem dos elementos de `s`.

   Apresente o resultado de aplicar essa função à stack apresentada como exemplo.

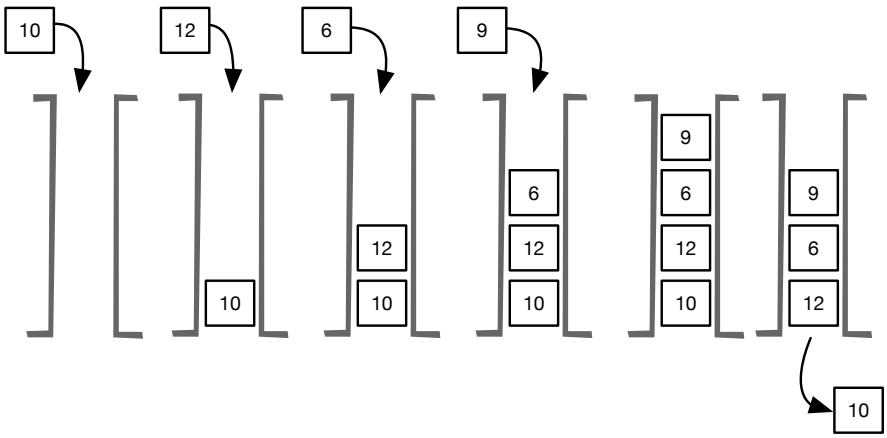
5. Apresente uma definição alternativa da função `reverse` da alínea anterior que reutiliza as células da lista, i.e., que não faz quaisquer `push` (`malloc`) ou `pop` (`free`).

3.2 Queues

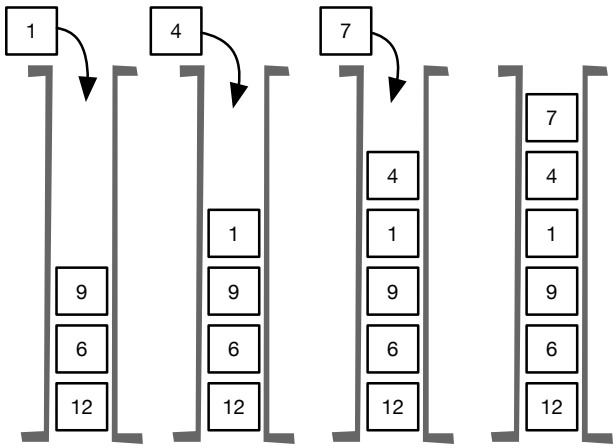
Uma queue é um buffer em que o próximo elemento a ser removido é o primeiro elemento inserido (*First In First Out*).
As operações de inserção e remoção de um elemento são normalmente referidas como **enqueue** e **dequeue**.

Exemplo 10 Vejamos então o resultado de, numa queue de inteiros, inicialmente vazia, se efecturem algumas destas operações.

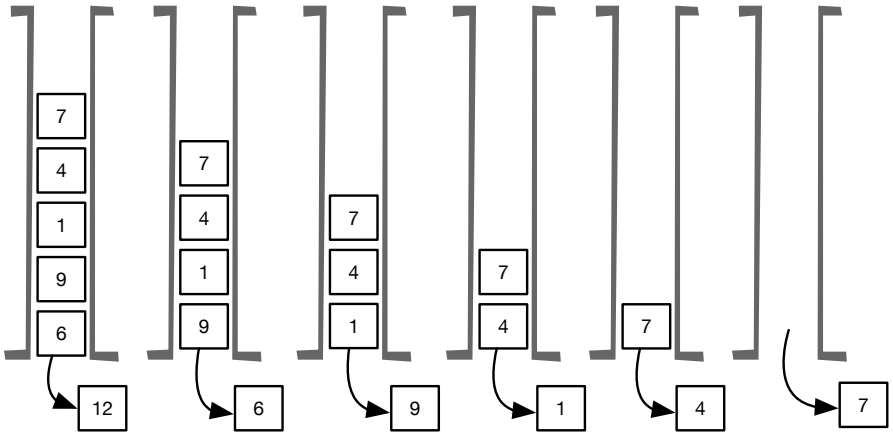
1. O resultado de acrescentar (`enqueue`) os números 10, 12, 6 e 9 é



2. Ao removermos (**dequeue**) um elemento o elemento removido é o 10
3. Após a inserção de 1 4 e 7 a queue passa a ter 6 elementos.



4. Para a *esvaziar* será necessário fazer 6 remoções. A ordem pela qual os elementos serão removidos é 12, 6, 9, 1, 4 e 7.



3.2.1 Usando arrays estáticos

A primeira implementação de queues que vamos apresentar usa um array para armazenar os elementos da queue. Relativamente à implementação de stacks há aqui uma dificuldade extra e que diz respeito à ordem pela qual os elementos são armazenados no array.

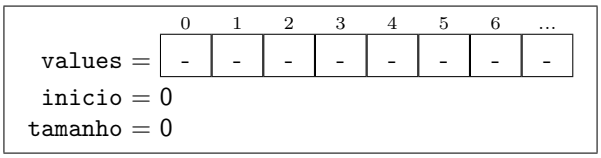
- Se optarmos por armazenar os elementos por ordem de entrada na queue, a operação de enqueue é bastante eficiente (trata-se de acrescentar um elemento no final do array), mas a operação de dequeue é bastante demorada (todos os elementos terão de ser ajustados uma posição).
- Se pelo contrário resolvermos armazenar os elementos por ordem de saída, aumentamos a eficiência da remoção mas voltamos a ter um problema similar com a operação de enqueue: todos os elementos terão que ser deslocados.

Para resolver este problema é comum usar, para além do tamanho da queue, o índice onde ela se inicia (i.e., onde está o próximo elemento a ser removido).

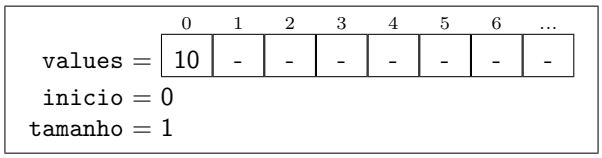
```
#define MAX 1000
typedef struct queue {
  int values [MAX];
  int inicio, tamanho;
} Queue;
```

Exemplo 11 Vejamos a evolução da queue apresentada atrás usando esta implementação.

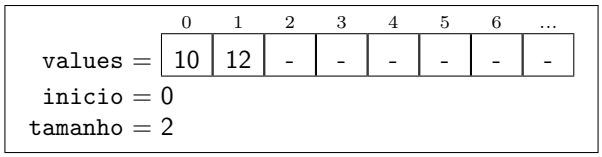
- queue vazia



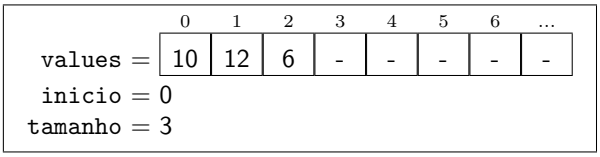
- `enqueue (10)`



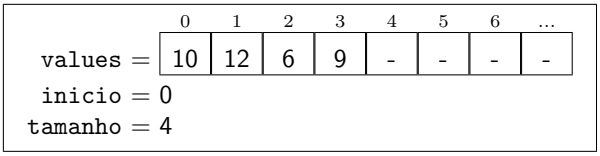
- `enqueue (12)`



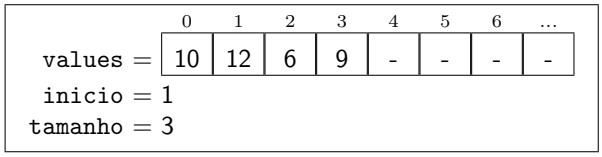
- `enqueue (6)`



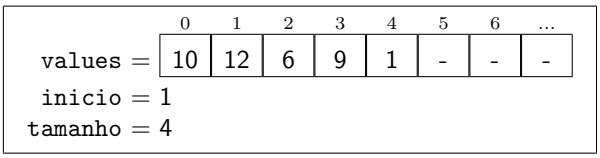
- `enqueue (9)`



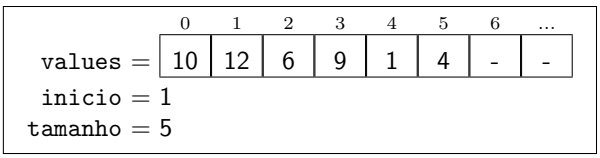
- `dequeue`



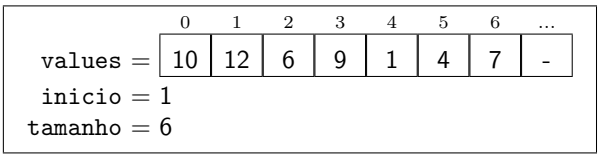
- `enqueue (1)`



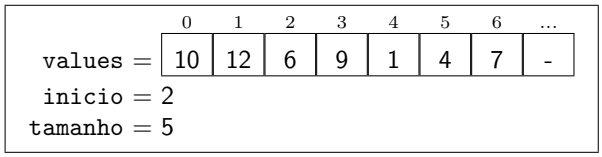
- `enqueue (4)`



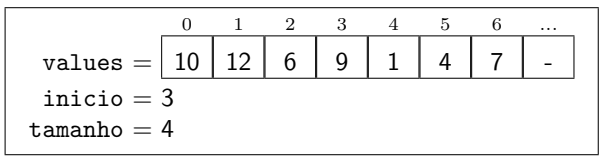
- `enqueue (7)`



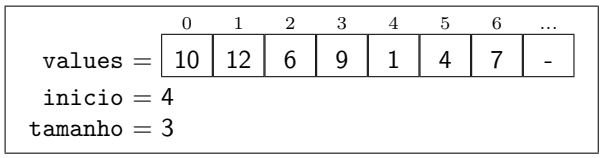
- `dequeue`



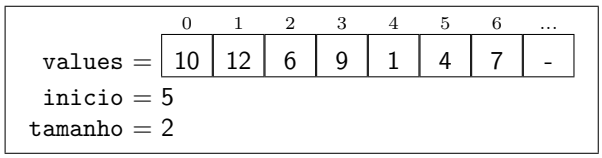
- `dequeue`



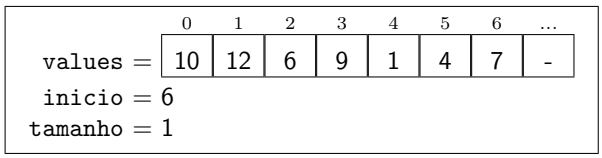
- dequeue



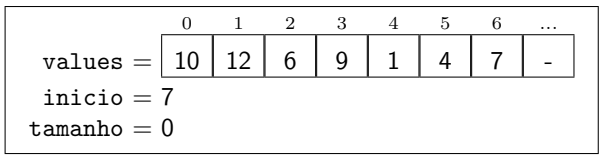
- dequeue



- dequeue



- dequeue



Os campos `inicio` e `tamanho` delimitam a zona do array `values` que é ocupada paela queue. Essa zona corresponde aos índices compreendidos entre `inicio` e `inicio + tamanho` Apesar deste exemplo não o evidenciar (por ser suficientemente pequeno), é conveniente que seja usada aritmética modular, i.e., o índice que se segue ao índice `MAX-1` é o índice `0`. Assim, por exemplo, para o valor de `MAX` fixado acima (`1000`), se `inicio = 997` e `tamanho = 5`, os elementos da queue estõa nos índices `997`, `998`, `999`, `0` e `1`, por esta ordem.
As funções de inicialização de uma queue e de teste se a queue está ou não vazia precisam apenas de manipular os campos `inicio` e `tamanho`.

```
void initQueue (Queue *q){
  q->inicio=0; q->tamanho=0;
}
```

```
int isEmpty (Queue *q){
  return (q->tamanho==0);
}
```

As operações de `enqueue` e `dequeue` retornam um código de sucesso uma vez que nem sempre são possíveis de efectuar. No caso da operação `enqueue`, isso acontece quando a queue esgotou a sua capacidade. No caso do `dequeue` quando a queue está vazia.

```
int enqueue (Queue *q, int x){
  int r=0;
  if (q->tamanho == MAX) r=1;
  else
    q->values[(q->inicio+q->tamanho++) % MAX] = x;
  return r;
}

int dequeue (Queue *q, int *x){
```

```
  int r=0;
  if (q->tamanho == 0) r=1;
  else {
    *x = q->values[q->inicio];
    q->inicio = (q->inicio + 1)%MAX;
    q->tamanho--;
  }
  return r;
}
```

3.2.2 Usando listas ligadas

Tal como acontecia com a implementação em array de stacka, esta implementação de queues limita à partida a capacidade das queues. Podemos ultrapassar esse problema armazenando os valores da queue numa lista ligada.
Neste caso a solução não é tão trivial uma vez que ambas as extremidades da lista terão de ser alteradas. Por isso é costume ter acesso directo a ambas as extremidades da lista.

- As inserções (enqueue) serão feitas no final da lista.
- As remoções são feitas no início da lista.

```
typedef struct celula {
  int value;
  struct celula *prox;
} Celula;

typedef struct queue{
  struct celula *inicio, *fim;
} Queue;
```

A queue vazia será representada por uma lista vazia. Ambos os endereços serão por isso inicializados a `NULL`.

```
void initQueue (Queue *q){
  q->inicio = q->fim = NULL;
}
```

```
int isEmpty (Queue q){
  return (q->inicio==NULL);
}
```

As operações de `enqueue` e `dequeue` retornam um código de sucesso uma vez que nem sempre são possíveis de efectuar. No caso da operação `enqueue`, isso acontece quando já não há memória disponível. No caso do `dequeue` quando a queue está vazia.

```
int enqueue (Queue *q, int x){
  int r=0;
  Celula *new;

  new=malloc(sizeof (Celula))
  if (new == NULL) r=1;
  else {
    new->value=x;
    new->prox=NULL;
    q->fim->prox=new;
    q->fim=new;
    if (q->inicio == NULL)
       q->inicio=new;
  }
  return r;
}
```

```
int dequeue (Queue *q, int *x){
  int r=0;
  Celula *tmp;
  if (q->inicio == NULL) r=1;
  else {
    *x = q->inicio->value;
    tmp=q->inicio;
    q->inicio=q->inicio->prox;
    if (q->inicio == NULL)
       q->fim=NULL;
    free (tmp);
  }
  return r;
}
```

Exercício 11 Uma forma de combinar as vantagens destas duas implementações consiste em usar arrays dinâmicos.

```
typedef struct queue {
  int size;
  int *values;
  int inicio, tamanho;
} Queue;
```

Defina as funções de manipulação de queues para esta implementação.
Note que, ao contrário do que acontecia com a implementação de stacks em arrays dinâmicos, a operação de duplicação do espaço deve ter em conta a circularidade do vector `values`, pelo que o uso da função `realloc` não permite por si só, fazer a dita duplicação.

3.3 Filas com prioridades

O último caso de buffers que vamos apresentar consiste num buffer em que o elemento que se retira é o menor dos que se encontram armazenados.
Para implementar um destes buffers podemos começar por analizar duas alternativas:

- Manter os elementos do buffer num array ordenado (por ordem decrescente). Desta forma, a remoção de um elemento é uma operação bastante eficiente.
- Manter os elementos do buffer num array armazenados por ordem de entrada no buffer. Desta forma a inserção de um novo elemento é uma operação bastante eficiente.

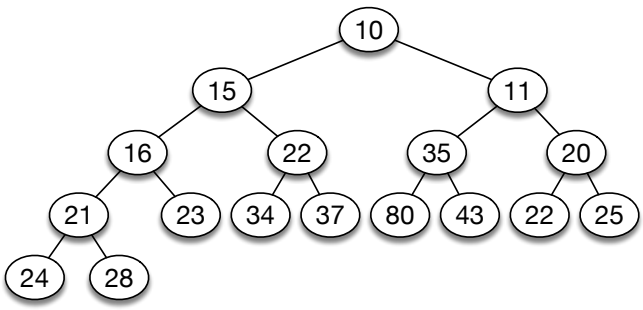
O problema com estas alternativas é que, sendo uma das operações (inserção ou remoção) eficientes a outra não é.

- Se os elementos do array estiverem por ordem descrescente, a inserção de um novo elemento pode obrigar a deslocar todos os elementos (no caso de o elemento a ser acrescentado ser maior do que os que já lá estão).
- Se os elementos estiverem armazenados por ordem de entrada no buffer, a operação de remoção terá obrigatoriamente de percorrer todo o buffer para determinar qual é o elemento a ser removido (mínimo).

Uma solução alternativa consiste em organizar o array onde os elementos estão armazenados numa **heap**, neste caso numa **min-heap**.
Uma min-heap é uma árvore binária em que cada elemento é menor ou igual aos seus sucessores.
Para implementar filas de prioridades, as min-heaps que são usadas obedecem ainda a uma propriedade extra sobre a forma da árvore em questão: deve-se tratar de árvores semi-completas, i.e., em que todas as folhas da árvore se encontram no último nível da árvore, ou à direita no nível anterior.
Esta última propriedade permite-nos, ao armazenar os elementos da árvore num array por níveis (i.e., guardando primeiro os elementos que estão no nível 1, depois os do nível 2, ...) determinar para o elemento que está no nível `i` que:

- o seu descendente esquerdo está no índice `2*i + 1`
- o seu descendente direito está no índice `2*i + 2`
- o seu ascendente está no nível `(i-1) / 2`

Exemplo 12 A min-heap

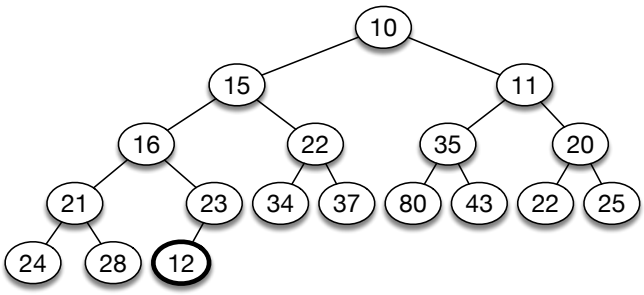


pode ser armazenada no array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	15	11	16	22	35	20	21	23	24	37	80	43	22	25	24	28

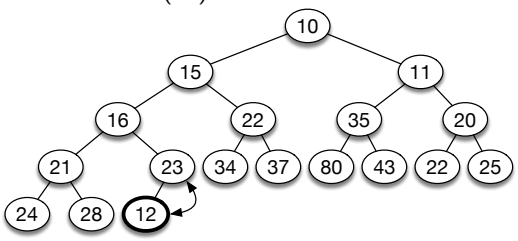
Vejamos agora como com esta implementação conseguimos que tanto a remoção do menor elemento como a adição de um novo elemento podem ser feitas em tempo proporcional ao logaritmo (na base 2) do número de elementos do buffer. Para isso convem recordar que, como a árvore que representa os elementos do buffer é semi-completa, a sua altura é no máximo o logaritmo do número de elementos. Basta-nos então mostrar que o número de iterações envolvidas tanto na operação de remoção como na de adição de um novo elemento correspondem a processos está limitada pela altura desta árvore.

Exemplo 13 Vejamos como acrescentar à min-heap anterior um novo valor, por exemplo o número 23.
O resultado desta operação tem que produzir uma min-heap, com as duas restrições que a definem: ordem e forma.
Se colocarmos o elemento a inserir no final do array (e consequentemente como elemento mais à direita do último nível da árvore) fica satisfeita a segunda das restrições (forma) mas não necessáriamente a primeira (ordem). A árvore resultante é:

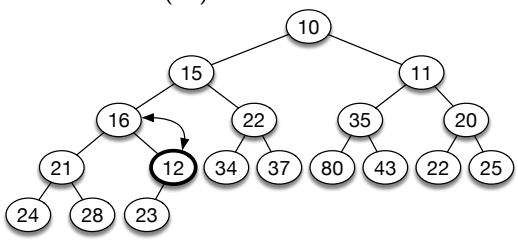


Para resolvermos este problema vamos *empurrando* o elemento inserido, por sucessivas trocas com o seu ascendente, até que esse elemento já não viole a propriedade de ordem, i.e., que o seu ascendente não seja maior.
Neste caso os passos a dar serão:

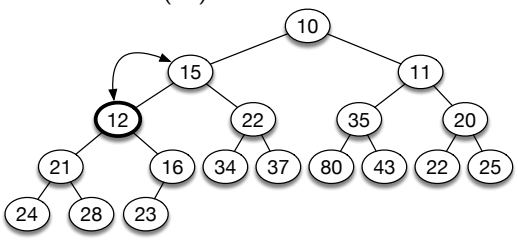
1: O elemento 12 tem que ser trocado com o seu antecessor (23)



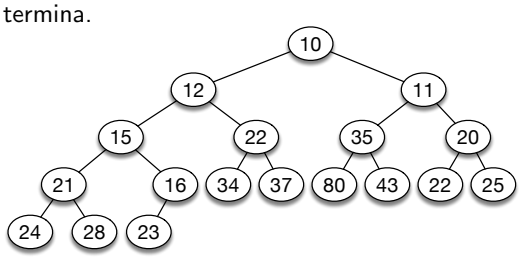
2: O elemento 12 tem que ser trocado com o seu antecessor (16)



3: O elemento 12 tem que ser trocado com o seu antecessor (15)



4: Como o antecessor de 12 já é menor ou igual a 12 a árvore já satisfaz a restrição de ordem e por isso trata-se de uma min-heap e o processo termina.



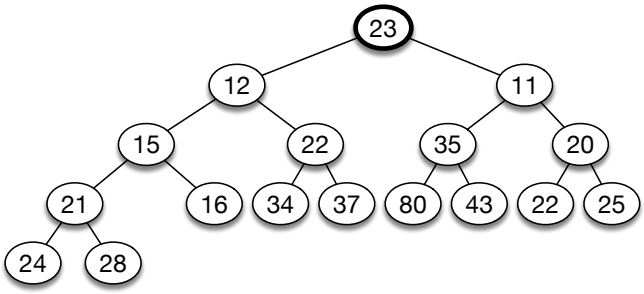
O procedimento exemplificado é normalmente conhecido por *bubble-up*.

```
void bubbleup (int v[], int i){
  int p = (i-1)/2; // antecessor de i
  while (i>0 && v[i] < v[p]){
```

```
    swap (v,i,p);
    i=p;              // nova posição do elemento
    p=(i-1)/2;        // antecessor de i
  }
}
```

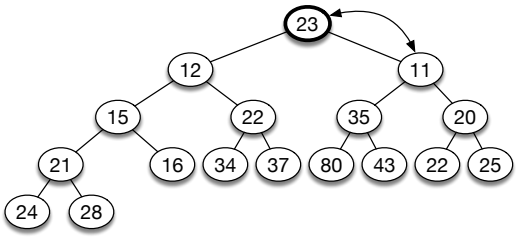
Note-se que, no pior caso (o caso em que o elemento inserido é menor do que todos os que lá estão), o número de iterações do ciclo acima está limitado por $\log_2(N)$ (em que N corresponde ao valor inicial de `i`): cada iteração do ciclo divide o valor de `i` por 2.

Exemplo 14 Vejamos agora a operação de remoção de um elemento (o menor).
Se colocarmos o último elemento no início do array (e consequentemente na raiz da árvore) fica satisfeita a segunda das restrições (forma) mas não a primeira (ordem). A árvore resultante é:

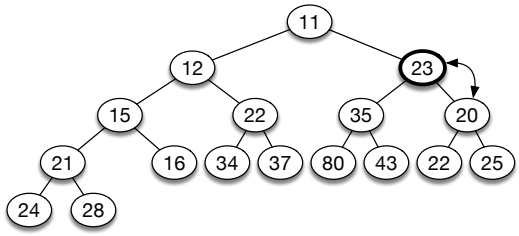


Para resolvermos este problema vamos *empurrando* o elemento inserido, por sucessivas trocas com o menor dos seus descendentes, até que esse elemento já não viole a propriedade de ordem, i.e., que seja menor do que ambos os descendentes.
Neste caso os passos a dar serão:

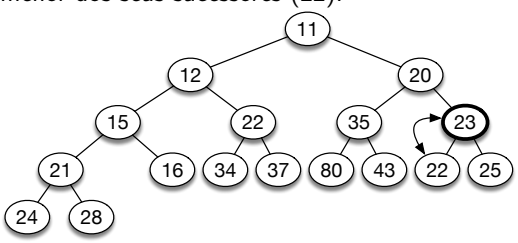
1: O elemento 23 tem que ser trocado com o menor dos seus sucessores (11).



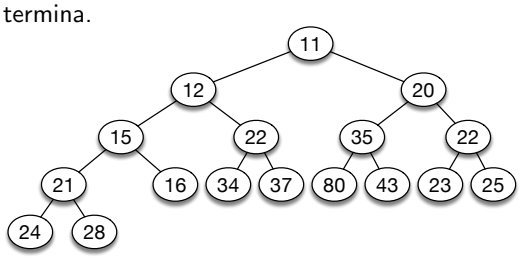
2: O elemento 23 tem que ser trocado com o menor dos seus sucessores (20).



3: O elemento 23 tem que ser trocado com o menor dos seus sucessores (22).



4: Como já não existem sucessores o processo termina.



O procedimento exemplificado é normalmente conhecido por *bubble-down*.

```
void bubbledown (int v[], int N, int i){
  int f = 2*i + 1;
  while (f<N) {
     if (f+1 < N && v[f+1] < v[f])
        // o descendente da direita é menor
        f = f+1;
     if (v[f]>v[i]) break;
     swap (v,i,f);
     i=f; f=2*i + 1;
  }
}
```

Mais uma vez podemos constatar que por cada iteração do ciclo o valor de `i` é multiplicado por 2, pelo que ao fim de no máximo $\log_2(N)$ iterações o ciclo terminará.
Usando estes dois procedimentos de manipulação de min-heaps estamos em condições de apresentar as definições das funções de manipulação de filas com prioridades.

```
#define MAX 1000
typedef struct prQueue {
  int values [MAX];
  int size;
} PriorityQ;
```

As funções de inicialização e de teste se está ou não vazia precisam apenas de manipular o campo `size`.

```
void initPriorityQ (PriorityQ *q){
  q->size=0;
}
```

```
int isEmpty (PriorityQ *q){
  return (q->size==0);
}
```

As operações de `add` e `remove` retornam um código de sucesso uma vez que nem sempre são possíveis de efectuar. No caso da operação `add`, isso acontece quando o buffer esgotou a sua capacidade. No caso do `remove` quando o buffer está vazio.

```
int add (PriorityQ *q, int x){
  int r=0;
  if (q->size == MAX) r=1;
  else {
    q->values[q->size] = x;
    bubbleup (q->values, q->size);
    q->size++;
  }
  return r;
}

int remove (PriorityQ *q, int *x){
  int r=0;
  if (q->size == 0) r=1;
```

```
  else {
    *x = s->values[0];
    s->values[0] = s->values [--s->size];
    bubbledown (s->values, s->size, 0);
  }
  return r;
}
```

Exercício 12

Considere a função `heapify` definida ao lado. Mostre a evolução do array `v` quando esta função é invocada com um array

`v={30,26,22,18,16,13,11,8,6,5,4,3}`

```
void heapify (int v[], int N) {
  int i;
  for (i=1;i<N;i++)
    bubbleup (v,i);
}
```

Exercício 13 O pior caso da execução da função `heapify` apresentada acima corresponde a um array ordenado por ordem decrescente. Nesse caso

- as duas primeiras chamadas a `bubbleup` envolvem uma iteração;
- as quatro seguintes envolvem duas iterações;
- as oito seguintes envolvem três iterações;

e assim sucessivamente. Com base nestas observações, analise a complexidade da função `heapify` apresentada.

Exercício 14 Considere a definição da função `heapSort` de ordenação de um vector de inteiros (por ordem decrescente).

Analise a complexidade desta função em termos do número de comparações efectuadas entre elementos do vector. Para isso comece por determinar o número de comparações efectuadas pelas operações auxiliares envolvidas: `bubbleup`, `bubbledown` e `heapify`.

```
void heapSort (int v[], int N) {
  heapify (v,N);
  while (--N > 0) {
     swap (v,0,N);
     bubbledown (v,N,0);
  }
}
```

Exercício 15 Considere a seguinte definição alternativa da função `heapify`
Note que a função `bubbledown` só é aplicada a metade dos elementos do array.
Além disso,

- para 1/4 dos elementos (a metade superior dessa metade) a função `bubbledown` faz no máximo 1 iteração.
- para 1/8 dos elementos a função `bubbledown` faz no máximo 2 iterações, e assim sucessivamente.

```
void heapify (int v[], int N){
  int i;
  for (i=(N-2)/2; i>=0; i--)
    bubbledown (v,N,i);
}
```

Com base nestas observações, analise a complexidade da função `heapify` apresentada.

4 Dicionários

Em informática é costume chamar-se **dicionário** (**array associativo** ou **função finita**) a um tipo de dados constituído por uma coleção de pares *(chave,informação)* em que nenhuma chave aparece repetida.
As operações que estão normalmente associadas a este tipo são:

- adição de um novo par,
- modificação da informação associada a uma chave,
- procura da informação associada a uma chave,
- remoção de um par.

4.1 Tabelas de Hash

A primeira das implementações que vamos referir – tabelas de hash – pode ser motivada pela implementação de multiconjuntos apresentadas atrás (secção 1).
Um multiconjunto é um dicionário: a cada elemento do multiconjunto (chave) está associada a sua multiplicidade (informação). A implementação apresentada usa como índices do array as chaves do dicionário. Tal como dissemos na altura, tal alternativa só é viável no caso de

- as chaves serem um subconjunto dos números naturais
- o tamanho do conjunto de partida (as chaves que existem no dicionário) ser semelhante ao tamanho do domínio (as chaves que podem existir no dicionário).

Quando qualquer destas restrições não se verificam podemos contornar o problema dividindo a correspondência (mapping) que queremos armazenar em duas correspondências:

- uma função – chamada **função de hash** – que *converte* cada chave num índice de um array.
- um array onde se armazena a informação necessária.

Desta forma, para aceder/procurar/modificar um par (k, i) começamos por usar a função aplicada à chave k de forma a determinar o índice do array. Calculado este índice, a informação nessa posição poderá ser acedida/modificada.
Há no entanto dois problemas associados a esta solução.

- É necessário conseguir determinar se uma dada chave pertence ou não ao conjunto de partida. Isso pode ser resolvido se cada posição do array contiver informação que indique se a informação que lá está armazenada é significativa.

- Como o tamanho do array é, em princípio, muito menor do que o domínio da correspondência, a função de conversão entre chaves e índices é necessáriamente não injectiva. Isso significa que chaves diferentes serão convertidas no mesmo índice. Para duas chaves convertidas no mesmo índice do array é por isso impossível determinar a qual delas corresponde a informação armazenada nessa posição do array.

Estas duas observações evidenciam que cada componente do array deve armazenar:

- Uma marca que sinaliza se essa componente do array está a ser usada.
- O par *(chave,informação)*.

Quando pretendemos armazenar dois pares (k_1, i_1) e (k_2, i_2) para os quais a função de hash aplicada a k_1 e k_2 coincide dizemos que ocorreu uma **colisão**. Veremos de seguida as várias alternativas para resolver este problema.

No código apresentado para exemplificar as várias implementações de tabelas de hash, vamos assumir que tanto as chaves como a informação associada são inteiros.
Vamos ainda assumir a existência de uma constante `HSIZE` que determina o tamanho do array e de uma função `int hash (int key, int size)` que converte uma chave (inteira neste caso) num índice do array (um número inteiro entre `0` e `size-1`)

4.1.1 Closed Addressing

Uma forma de resolver colisões é guardar em cada posição do array todos os pares *(chave,informação)* que colidem nessa posição.
Esse método é conhecido por **chaining** ou **closed addressing**.
Esses pares podem ser armazenados por exemplo numa lista ligada.

```
typedef struct bucket {
   int key; int info;
   struct bucket *next;
} *Bucket;

typedef Bucket HashTableChain [HSIZE];
```

As operações sobre tabelas de hash baseiam-se nas correspondentes operações sobre listas ligadas.

```
void initTab (HashTableChain h) {
  int i;

  for (i=0; i<HSIZE; h[i++]=NULL);
}

int lookup (HashTableChain h, int k, int *i){
    int p = hash (k,HSIZE);
    int found;
```

```
    Bucket it;

    for (it = h[p];
         it!=NULL && it->key!=k;
         it=it->next)
      ;
    if (it!=NULL) {
       *i=it->info;
       found = 1;
    }
    else found = 0;
    return found;
}

int update (HashTableChain h, int k, int i){
    int p = hash (k,HSIZE);
    int new;
    Bucket it;

    for (it = h[p];
         it!=NULL && it->key!=k;
         it=it->next)
      ;
    if (it!=NULL) {
       it->info = i;
       new = 0;
    }
    else {
       it = (Bucket) malloc (sizeof (struct bucket));
       it->info = i;
       it->key = k;
       it->next=h[p];
       h[p]=it;
       new = 1;
    }
    return new;
}

int removeKey (HashTableChain h, int k){
    int p = hash (k,HSIZE);
    int removed=0;
    Bucket *it, tmp;

    for (it = h+p;
         *it!=NULL && (*it)->key!=k;
         it=&((*it)->next))
```

```
        ;
    if (*it!=NULL) {
        tmp = *it;
        *it = (*it)->next;
        free (tmp);
        removed = 1;
    }
    return removed;
}
```

Esta implementação de tabelas de hash tem como grande vantagem a sua simplicidade. Tem ainda um bom comportamento desde que as listas de colisões não cresçam demasiado.
Uma medida habitualmente associada às tabelas de hash é o **factor de carga** que se defini como a razão entre o número de pares armazenados e o tamanho do array que está a ser usado

$$\lambda = \frac{\text{nº de chaves}}{\text{tamanho do array}}$$

No caso desta implementação (closed addressing), o factor de carga mede o tamanho médio de cada *bucket*, i.e., o número médio de colisões que aconteceram na tabela.
Esta medida acaba por corresponder ao tempo médio de acesso a uma chave e por isso deve ser mantido o mais baixo possível.
Mais uma vez, uma alternativa para corrigir potenciais desajustes, consiste em usar um array dinâmico em vez de estáticamente definido em tempo de compilação.
Convém no entanto referir que a operação de duplicação do tamanho do vector precisa de re-inserir todos os pares uma vez que o valor do hash de uma dada chave depende do tamanho do array onde ela vai ser inserida: se mudarmos este tamanho é natural que esse valor, e consequentemente o *bucket* onde esse par será armazenado, mude.

Exercício 16 Considere a segunte definição para implementar tabelas de hash em closed addressin (chaining) usando um array dinâmico.

```
typedef struct bucket {
   int key; int info;
   struct bucket *next;
} *Bucket;
```

```
typedef struct thash {
    int size;
    Bucket *tabela;
} *THash;
```

1. Defina a função `THash createTable (int N)` que cria uma tabela de hash com um array de dimensão `N` inicialmente vazia.

2. Defina a função `int duplicateTable (THash t)` que duplica o tamanho do array de uma dada tabela. Assuma que existe definida a função `int hash (int key, int size)` que calcula o hash de uma chave `key`.

4.1.2 Open Addressing

Uma outra forma de lidar com colisões consiste em manter todos os pares *(chave,informação)* num único array. A posição em que um dado par deverá ser armazenado/consultado é calculada usando a função de hash. No caso de acontecer uma colisão, é calculada uma nova posição para armazenar esse par. Este processo de calcular uma posição alternativa é conhecido por **probing** e pode ter várias soluções.

Vamos começar por apresentar a mais simples destas alternativas, conhecida por **linear probing** e que consiste no seguinte. No caso de querermos guardar um par numa dada posição `p` já ocupada, tentamos fazê-lo na posição seguinte. Se esta também estiver ocupada repetimos o processo até que seja atingida uma posição não ocupada.
Vejamos então como codificar esta solução, ignorando para já a remoção de um par.
Mais uma vez, assumimos a existência de uma constante `HSIZE` que determina o tamanho do array e de uma função `int hash (int key, int size)` que converte uma chave (inteira neste caso) num índice do array (um número inteiro entre `0` e `size-1`).

```
#define STATUS_FREE 0
#define STATUS_USED 1

typedef struct bucket {
   int status;
   int key; int info;
} Bucket;

typedef Bucket HashTable [HSIZE];
```

A função de inicialização deverá marcar todas as posições do array como não ocupadas.

```
void initTab (HashTable h) {
  int i;

  for (i=0; i<HSIZE; h[i++].status=STATUS_FREE);
}
```

Antes de apresentarmos as definições das funções de procura e inserção de uma nova chave vamos definir uma função que, dada uma chave, determina a posição do array onde essa chave deveria ser armazenada.
A definição desta função depende do método de probing que é utilizado e, tal como dissemos atrás, vamos começar por apresentar a mais simples dessas estratégias – linear probing.

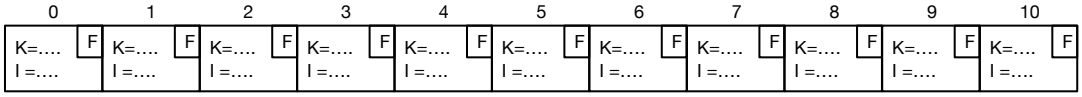
```
int find_probe (HashTable h, int k) {
    int p = hash (k,HSIZE);
    int count;

    for (count=HSIZE;
         count>0
        && h[p].status != STATUS_FREE
```

```
          && h[p].key      != k;
           count--)
       p=(p+1)%HSIZE;
    if (count == 0)
       // table is full
       p=-1;
    return p;
}
```

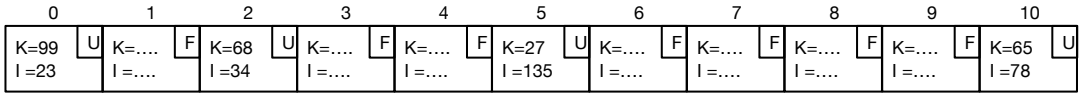
Note-se que o array usado é considerado *circular*, i.e., que a posição seguinte à posição `HSIZE-1` é a posição `0`.
Usando esta função podemos definir as funções de consulta e de inserção de um novo par.

```
int lookup (HashTable h, int k, int *i){
    int p = find_probe (h,k);
    int found;

    if (p>=0 && h[p].key==k) {
       *i=h[p].info;
       found = 1;
    }
    else found = 0;

    return found;
}

int update (HashTable h, int k, int i){
    int p = find_probe (h,k);
    int r;

    if (p<0) // table is full
       r=0;
    else if (h[p].key==k) {
       // key exists
       h[p].info = i;
       r = 1;
    }
    else { // new key
       h[p].status = STATUS_USED;
       h[p].key=k;
       h[p].info=i;
       r = 2;
    }
    return r;
}
```

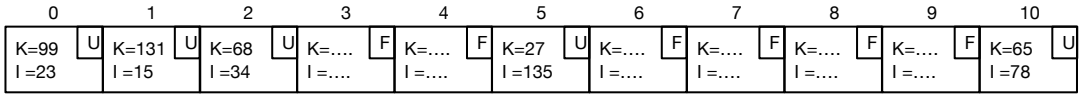
Exemplo 15 Para ilustrar o comportamento das funções apresentadas acima vamos apresentar a evolução de uma tabela de hash (com tamanho `HSIZE=11`) após a invocação destes procedimentos. Suponhamos que a função de hash é definida como `hash (k,s) = k%s`
Vamos partir de uma tabela inicialmente vazia.



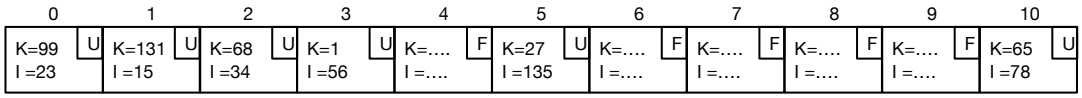
A inserção dos pares `(27,135)`, `(68,34)`, `(65,78)` e `(99,23)` não provoca nenhuma colisão: os valores da função de hash para estes pares são, respectivamente, 5, 2, 10 e 0 e correspondem a posições livres do array.
Após essas inserções o array estará no seguinte estado.



A inserção do par (131,15) (note-se que a `131%11=10`) vai colocá-lo na posição 1 (10 está ocupada e a posição seguinte – 0 – também).



A inserção do par `(1,56)` vai obrigar este par a ser inserido na posição 3 do array.



Se neste estado consultarmos (`lookup`) as chaves 99 e 44 (ambas com hash 0) conseguimos recuperar a informação da primeira e concluir que a segunda não se encontra armazenada (a função `find_probe` acima dá como resultado a posição 4 que está livre).

Note-se que as colisões que aconteceram na última inserção deste exemplo são todas com chaves cujo hash é diferente do da chave original. Estas colisões são conhecidas como **colisões secundárias** e são uma consequência do método de tratar colisões adoptado (open addressing). A mesma sequência de inserções numa tabela com closed addressing não produziria colisões secundárias.
As operações apresentadas acima baseiam-se todas na função `find_probe` onde está realmente implementada a estratégia de **linear probing**. Esta estratégia tem como principal desvantagem não evitar que se criem na tabela *clusters* de células ocupadas aumentando por isso o número de colisões secundárias. Uma forma de evitar isto consiste em ir aumentando o *espaçamento* entre probes sucessivos. Assim, em vez de tentarmos na posição imediatamente seguinte, vamos procurar `d` posições à frente, incrementando esta medida `d`.
Esta estratégia é conhecida como **quadratic probing**.

```
int find_Quad_probe (HashTable h, int k) {
    int p = hash (k,HSIZE);
    int count, d=1;

    for (count=HSIZE;
         count>0
```

```
            && h[p].status != STATUS_FREE
            && h[p].key    != k;
            count--) {
        p=(p+d)%HSIZE; d=d+1;
    }
    if (count == 0)
        // table is full
        p=-1;
    return p;
}
```

A remoção de uma chave em tabelas de hash que usam open-addressing não pode ser feita simplesmente passando e estado da correspondente componente do array a livre.

Exemplo 15 (continuação) Se, no ponto em que a tabela se encontra, removermos a chave 131 (na posição 1, apenas passando o estado dessa componente para livre, uma procura da chave 1 passaria a não conseguir descobrir que essa chave existe de facto na tabela.

A forma mais usual de efectuar remoções numa destas tabelas consiste em usar um valor adicional no campo `status` para marcar que essa entrada foi apagada.
As procuras e inserções terão que ter isso em conta.

```
#define STATUS_FREE    0
#define STATUS_USED    1
#define STATUS_DELETED 2
```

Em primeiro lugar, relembremos o ciclo (de probing) definido na função `find_probe` da página 41.

```
    for (count=HSIZE;
         count>0
         && h[p].status != STATUS_FREE
         && h[p].key    != k;
         count--)
        p=(p+1)%HSIZE;
```

Admitindo a existência de mais um valor para o estado de cada componente (`STATUS_DELETED`) este ciclo trata tal estado como se se tratasse de uma componente ocupada.
Utilizar esta definição significa que as componentes apagadas nunca serão reaproveitadas para armazenar outra informação.
Vamos então redefinir esta função de forma a que:

- Se a chave existir (apagada ou efectiva) a função retorna a posição correspondente.
- Se a chave não existir a função retorna a primeira posição livre ou ocupada *a partir* da posição dada pela função de hash.

```
int find_probe (HashTable h, int k) {
    int p = hash (k,HSIZE), tmp;
    int count;

    for (count=HSIZE;
         count>0
         && h[p].status == STATUS_USED
         && h[p].key     != k;
         count--)
      p=(p+1)%HSIZE;
    if (h[p].key != k)
       if (count == 0)
          // table is full
          p=-1;
       else { // deleted cell found
          tmp = p;
          while ( count>0
                  && h[p].status != STATUS_FREE
                  && h[p].key     != k ) {
              count--; p=(p+1)%HSIZE;
          }
          if (h[p].key != k) p=tmp;
       }
    return p;
}
```

Apesar de o código acima ser substancialmente mais extenso do que a primeira definição, convem notar que uma remoção não deteriora a eficiência das pesquisas/inserções. A pesquisa de uma qualquer chave não passará a ser menos eficiente após uma remoção. Em termos práticos as células removidas são tratadas como se de células efectivas se tratassem.
Vejamos então como implementar a operação de remoção de uma chave bem como as funções de inserção e consulta.

```
int lookup (HashTable h, int k, int *i){
    int p = find_probe (h,k);
    int found;

    if (p>=0
        && h[p].key==k
        && h[p].status==STATUS_USED) {
       *i=h[p].info;
       found = 1;
    }
    else found = 0;

    return found;
```

```
}

int update (HashTable h, int k, int i){
    int p = find_probe (h,k);
    int r;

    if (p<0) // table is full
       r=0;
    else if (h[p].key==k) {
       // key exists
       h[p].info = i;
       h[p].status=STATUS_USED
       r = 1;
    }
    else { // new key
       h[p].status = STATUS_USED;
       h[p].key=k;
       h[p].info=i;
       r = 2;
    }
    return r;
}

int delete (HashTable h, int k){
    int p = find_probe (h,k);
    int r=0;

    if (p>0
        && h[p].key==k
        && h[p].status==STATUS_USED)
         // key exists
            h[p].status=STATUS_DELETED;
    else // key does not exist
            r = 1;
    }
    return r;
}
```

Exercício 17 Considere o estado final da tabela de hash apresentada no exemplo 15. Mostre a evolução da tabela após as seguintes operações.

1. remoção das chaves 99 e 131.
2. inserção dos pares `(121,44)` e `(67,66)`.
3. remoção da chave 56.

Exercício 18 Considere que se fizeram exactamente a mesma sequência de operações descritas no Exemplo 15 e no exercício anterior, mas sobre uma tabela de hash com o mesmo tamanho (HSIZE=11) implementada com chaining. Apresente a evolução dessa tabela.

O tratamento de colisões usando open addressing tem um comportamento aceitável desde que o factor de carga seja relativamente baixo. Além disso, e como vimos nas definições acima, as células apagadas são tratadas como se de células efectivas fossem. Face a estas considerações, associada a esta técnica de tratamento de colisões, é habitual:

- Implementar as tabelas usando arrays dinâmicos que podem ser redimensionados quando o factor de carga ultrapassa um determinado valor.
- Definir um procedimento de limpeza das células apagadas (*garbage collection*).

Qualquer um destes dois procedimentos terá que percorrer todo o array para re-inserir os pares efectivos numa tabela inicialmente vazia. Esta re-inserção é uma versão simplificada do que foi apresentado acima pois não terá que prever os casos de a tabela estar cheia nem de existirem células apagadas.

```
void rehash (int S, Bucket source[S],
             int T, Bucket target[T]) {
     int is, it;
     // clean target
     for (it=0; it<T; it ++)
       target[it].status = STATUS_FREE;
     // traverse the source
     for (is=0; is<S; is++)
        if (source[is].status==STATUS_USED) {
          it = hash (source[is].key,T);
          // probe
          while (target[it].status!=STATUS_FREE)
             it=(it+1)%T;
          // fill in the target
          target[it].status=STATUS_USED;
          target[it].key=source[is].key;
          target[it].info=source[is].info;
        }
}
```

Exercício 19 Usando a função `rehash` acima, defina uma função `void garbageC (HashTable h)` que remove da tabela `h` todas as células marcadas a apagadas.

Exercício 20 Considere a definição abaixo para implementar tabelas de hash com open addressing usando arrays dinâmicos.
Usando a função `rehash` acima, defina uma função `void doubleTable (HashTable *h)` que duplica o tamanho do array usado por uma tabela de hash.

```
typedef struct bucket {
   int status;
   int key; int info;
} Bucket;
```

```
typedef struct hashtable {
   int size;
   int used;
   Bucket *Table;
} HashTable;
```

Vamos terminar esta apresentação sobre implementação de tabelas de hash com open-addressing apresentando uma variante em que, para as células efectivas do array guarda o número de *probes* que foram efectuados para guardar essa informação nessa posição. Como este número é um número não negativo, podemos ainda usar quaisquer dois números negativos para marcar células livres (`-1`) e apagadas (`-2`).

```
typedef struct bucket {
  int probeC; // -1: Free; -2: Deleted
  int key;
  int value;
} Bucket;

typedef Bucket HashTable [HSIZE];
```

As funções de actualização (`update`) e consulta (`lookuop`) são em tudo semelhantes ao que fizemos atrás.

```
int update (HashTable t, int key, int value) {
  int i=hash (key,HSIZE), probe = 0;

  while ((t[i].probeC >= 0) &&
        (t[i].key != key) &&
        (probe < HSIZE)) {
    i=(i+1)%HSIZE; probe++;
  }
  if (probe > HSIZE) return 2; // overload
  if (t[i].probeC != FREE) {
     t[i].value = value;
     return 1; // exists
  }
  t[i].key = key;
  t[i].value = value;
  t[i].probeC = probe;
  return 0; // newkey
}

int lookup (HashTable t, int key, int *value) {
  int i=hash (key,HSIZE), probe=0;

  while ((t[i].probeC >= 0) &&
```

```
          (t[i].key != key) &&
          (probe < HSIZE)) {
      i=(i+1)%HSIZE; probe++;
  }
  if (probe == HSIZE) return 2; // overload
  if (t[i].probeC >= 0) {
     *value = t[i].value;
     return 0; // exists
  }
  return 1; // fail
}
```

Esta pequena modificação na representação permite determinar facilmente quando uma colisão é secundária ou não. Este facto vai-nos permitir implementar uma função de remoção efectiva de uma chave (i.e., sem a marcar apenas como removida).

Exercício 21 Defina uma função `int count (HashTable t, int i)` que, dada uma tabela de hash `t` e uma posição `i`, determina quantas das chaves armazenadas na tabela têm hash `i`. A função não deve percorrer necessariamente toda a tabela nem calcular o hash de qualquer das chaves armazenadas.

Exercício 22 Defina uma função `int last (HashTable t, int i)` que calcula a posição da *última* chave armazenada na tabela com hash `i`.
A função deverá retornar `-1` caso tal chave não exista e não deve percorrer necessariamente toda a tabela nem calcular o hash de qualquer das chaves armazenadas.

Exercício 23 Defina uma função `int lastP (HashTable t, int i)` que calcula a posição do *último* elemento que poderia ter sido armazenado na posição `i` se estivesse livre.
A função deverá retornar `-1` caso tal chave não exista e não deve percorrer necessariamente toda a tabela nem calcular o hash de qualquer das chaves armazenadas.

Ainda falta remoção usando implementações com probe count

4.2 Árvores de procura balanceadas

As tabelas de hash são uma forma simples de implementar um dicionário mas têm algumas desvantagens.
Em primeiro lugar, apesar de, se bem dimensionadas, e com uma boa função de hash, terem um comportamento esperado óptimo (as operações de consulta e inserção/actualização executam em tempo constante) este comportamento pode degenerar e uma análise do pior caso mostra que essas operações executam nesses casos em tempo linear (no tamanho do dicionário). Em segundo lugar, e como não existe qualquer exigência de monotonia na função de hash, a informação é armazenada no array por uma ordem que não a ordem natural das chaves. Isto faz com que as tabelas de hash não sejam indicadas para problemas em que seja necessário percorrer todo o dicionário por ordem das chaves.
Uma alternativa para armazenar um dicionário onde existe uma relação de ordem entre as chaves são árvores binárias de procura (ordenadas pela chave).

```
typedef struct bst {
  int key, info;
  struct bst *left, *right;
} *BST;
```

As operações de inserção e consulta são guiadas pela relação entre a chave a inserir/-procurar e a que está no presente nodo.

```
int update (BST *a, int k, int i){
  int u = 0;

  while (*a != NULL && (*a)->key!=k)
     if ((*a)->key > k) a = &((*a)->left);
     else a = &((*a)->right);

  if (*a == NULL) {
    *a = (BST) malloc (sizeof (struct bst));
    (*a)->key=k;
    (*a)->info=i;
    (*a)->left=(*a)->right=NULL;
    u=1;
  } else
    (*a)->info = i;
  return u;
}
```

```
int lookup (BST a, int k, int *i){
  int found = 1;
  while (a!=NULL && a->key != k)
     if (a->key > k) a = a->left;
     else a = a->right;
  if (a!=NULL) *i=a->info;
  else found=0;
  return found;
}
```

Em cada iteração dos ciclos consulta-se um nível diferente da árvore. Por isso, ambas as funções têm um pior caso (em termos de complexidade) que corresponde à altura da árvore. Isto significa que, num dicionário com N pares, este pior caso pode variar entre N (se a árvore degenerar numa lista) e $\log_2(N)$ se se tratar de uma árvore perfeitamente equilibrada.

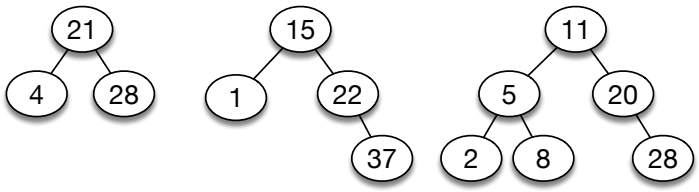
Embora o caso médio continue a corresponder a ter uma árvore com uma altura proporcional ao logaritmo do número de nodos existem formas de, sem prejudicar a eficiência da operação de inserção de uma nova chava, garantir que a altura da árvore permanece proporcional ao logaritmo do número de nodos.

Uma árvore binária diz-se **equilibrada** sse ou é vazia ou

- a diferença entre o número de nodos das suas duas sub-árvores é menor ou igual a 1 e

- ambas as sub-árvores são equilibradas.

Exemplo 16 As árvores



são equilibradas.

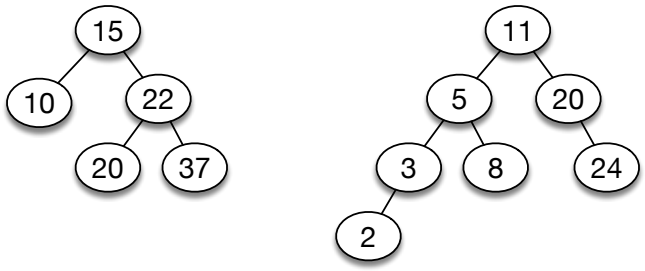
Uma árvore equilibrada com $N > 0$ nodos tem altura

$$1 + \lfloor \log_2(N) \rfloor = \mathcal{O}(\log_2(N))$$

Uma propriedade menos exigente das árvores mas que mantem esta última propriedade (de a altura ser proporcional ao logaritmo do número de elementos) é o **balanceamento**. Uma árvore binária diz-se **balanceada** sse é vazia ou

- a diferença entre as alturas das suas duas sub-árvores é menor ou igual a 1 e
- ambas as sub-árvores são balanceadas.

Exemplo 17 As árvores



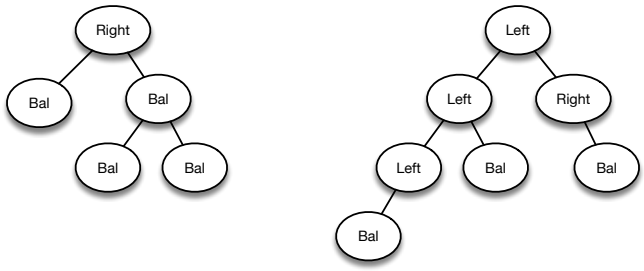
são balanceadas (e nenhuma delas é equilibrada).

De seguida vamos apresentar um algoritmo de inserção (conhecido como algoritmo de inserção em **árvores AVL**) em árvores binárias de procura que preserva o balanceamento, e apesar disso continua a ter um comportamento logarítmico no número de nodos da árvore.
De forma a atingir esse objectivo vamos guardar em cada nodo da árvore um novo campo – `bal` – que traduz o actual estado de balanceamento da árvore que aí se inicia. Esse campo poderá ter um de três valores que correspondem á diferença entre as alturas da sub-ávore esquerda e da direita

- `#define LEFT 1` quando a sub-árvore da esquerda é mais alta que a da direita.
- `#define BAL 0` quando as duas subárvores têm a mesma altura.

- `#define RIGHT -1` quando a sub-árvore da direita é mais alta que a da esquerda.

Exemplo 17 (continuação) Revisitando as árvores do exemplo anterior, apresentando apenas os factores de balanço de cada nó, teremos

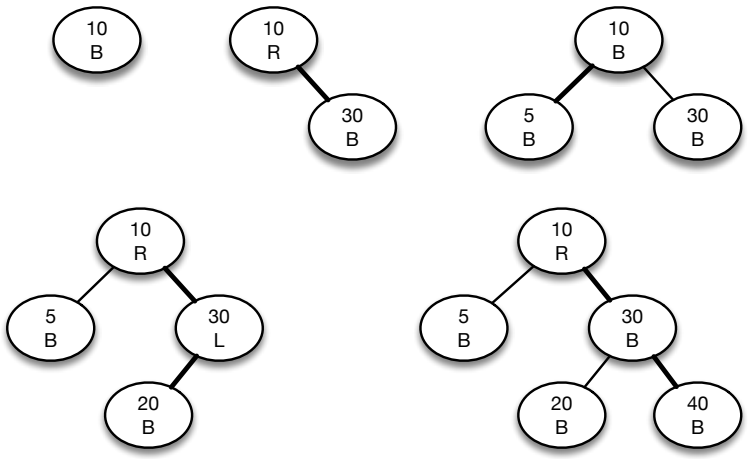



```
typedef struct avl {
  int bal;
  int key, info;
  struct avl *left, *right;
} *AVLTree;
```

Tal como acontecia com a inserção em árvores dinárias de procura, cada novo par vai ser sempre inserido como uma nova folha. Ao contrário do que acontecia nesse caso vamos ter que definir este procedimento recursivamente pois os factores de balanço terão que ser eventualmente recalculados. Também por esse motivo é necessário que cada inserção indique se como resultado dessa inserção a árvore em questão aumentou a sua altura.

```
int updateAVL (AVLTree *a, int k, int i){

  int g, u;
  *a = updateAVLRec (*a, k, i, &g, &u);
  return u;
}
```

Exemplo 18 Antes de prosseguirmos com a apresentação do algoritmo de inserção vejamos o que acontece na inserção, numa árvore inicialmente vazia, dos elementos 10, 30, 5, 20 e 40, por esta ordem.



É de notar que os factores de balanço que são eventualmente actualizados são os dos nodos que ligam o novo nodo (folha) à raiz da árvore.

Para melhor compreender o algoritmo de inserção convem relembrar os pressupostos deste algoritmo.

- A árvore recebida como argumento está balanceada.
- A árvore produzida está balanceada.

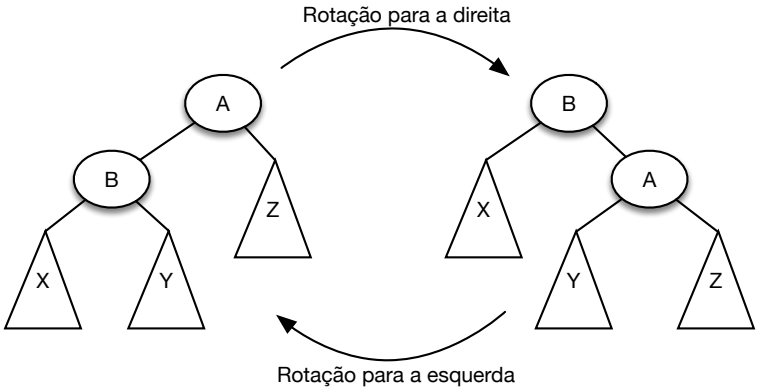
Após fazer uma chamada recursiva (e consequentemente receber como resultado uma árvore balanceada e a informação se essa árvore tem ou não uma altura superior à árvore original) devem ser feitos os ajustes necessários aos factores de balanço. Estes ajustes só são necessários se a árvore onde foi feita a inserção aumentar a sua altura.
Podemos enumerar então os vários casos que podem acontecer. Comecemos por apresentar os casos em que a árvore permanece balanceada sem qualquer acção adicional.

1. O factor de balanço é `BAL` e a inserção ocorreu na sub-árvore esquerda. Neste caso o factor de balanço deverá ser alterado para `LEFT`. A árvore aumenta de altura.
2. O factor de balanço é `RIGHT` e a inserção ocorreu na sub-árvore esquerda. Neste caso o factor de balanço deverá ser alterado para `BAL`. A árvore **não** aumenta de altura.
3. O factor de balanço é `BAL` e a inserção ocorreu na sub-árvore direita. Neste caso o factor de balanço deverá ser alterado para `RIGHT`. A árvore aumenta de altura.
4. O factor de balanço é `LEFT` e a inserção ocorreu na sub-árvore direita. Neste caso o factor de balanço deverá ser alterado para `BAL`. A árvore **não** aumenta de altura.

Nos casos restantes é preciso re-ajustar a forma da árvore.

5. O factor de balanço é `RIGHT` e a inserção ocorreu na sub-árvore direita. Neste caso a árvore ficou desbalanceada para a direita (a diferença de alturas passou a ser -2).
6. O factor de balanço é `LEFT` e a inserção ocorreu na sub-árvore esquerda. Neste caso a árvore ficou desbalanceada para a direita (a diferença de alturas passou a ser 2).

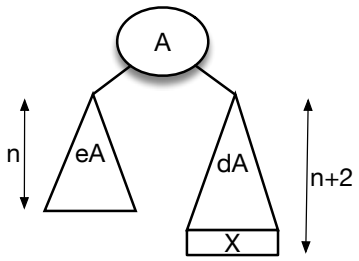
As operações elementares de re-ajuste de árvores que vamos usar são conhecidas como **rotações** e têm como principal característica preservar a ordem da árvore.




```
AVLTree rotateRight (AVLTree a){
    AVLTree b =  a->left;
    a->left = b->right;
    b->right = a;
    return b;
}
```

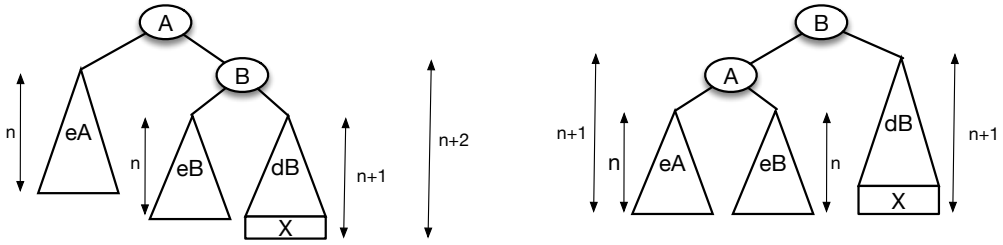
```
AVLTree rotateLeft (AVLTree b){
    AVLTree a = b->right;
    b->right = a->left;
    a->left = b;
    return a;
}
```

Vejamos então como tratar os casos em que a árvore terá que ser re-ajustada. Esses casos são perfeitamente simétricos e por isso analisaremos em pormenor apenas um deles.
Vamos então assumir que numa árvore cujo factor de balanço é `RIGHT` se faz uma inserção na sub-árvore direita donde resulta uma árvore balanceada cuja altura aumentou.
Convem notar que

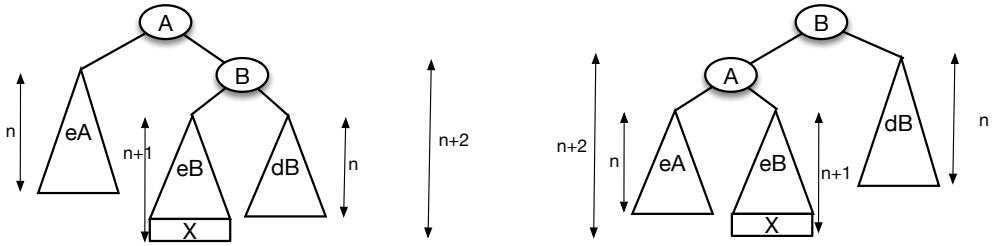


- essa sub-árvore antes da inserção (`dA`) era balanceada e assim permanece após a inserção.
- como a diferença de alturas com a sub-árvore esquerda é 2, antes da inserção essa árvore não era vazia.
- o factor de balanço desta sub-árvore só pode ser `LEFT` ou `RIGHT`.

Analisemos então os dois casos possíveis para o factor de balanço desta sub-árvore.
No caso deste ser `RIGHT` uma rotação para a esquerda da árvore corrige o desbalanceamento. Na figura abaixo podemos ver como os factores de balanço devem ser recalculados. Podemos ainda constatar que este re-ajuste faz com que a árvore, após a inserção do novo elemento, não aumenta de altura.



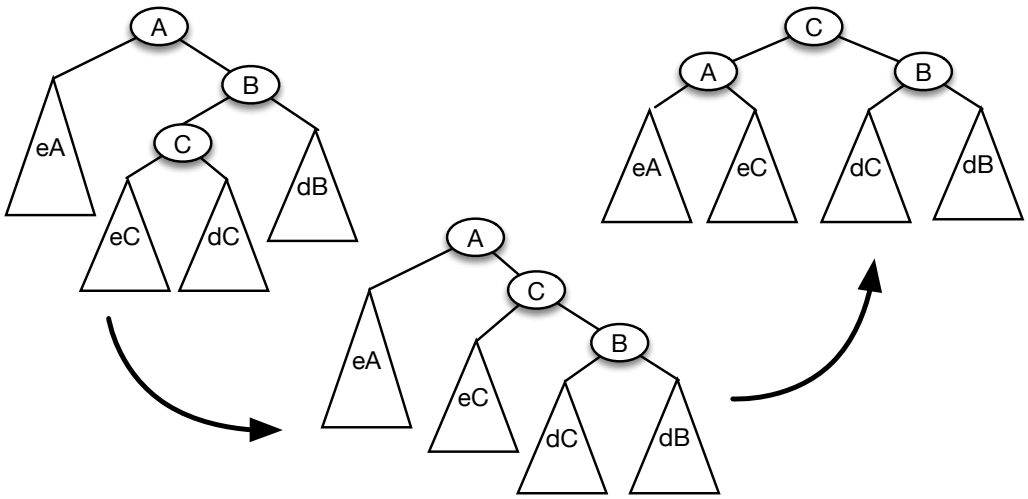
Vejamos agora como o mesmo reajuste não pode ser usado para rebaçancear a árvore no caso do factor de balanço ser `LEFT`.



É de notar que a sub-árvore esquerda da sub-árvore direita (`eB` na figura acima) é não vazia: tem pelo menos o elemento que acabou de ser inserido.
Para resolver este caso vamos ter que fazer um reajuste diferente e que consiste em fazer

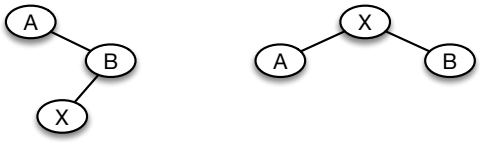
1. uma rotação para a direita na sub-árvore direita, seguida de
2. uma rotação para a esquerda na árvore resultante.

Vejamos então como tal transforma a árvore.

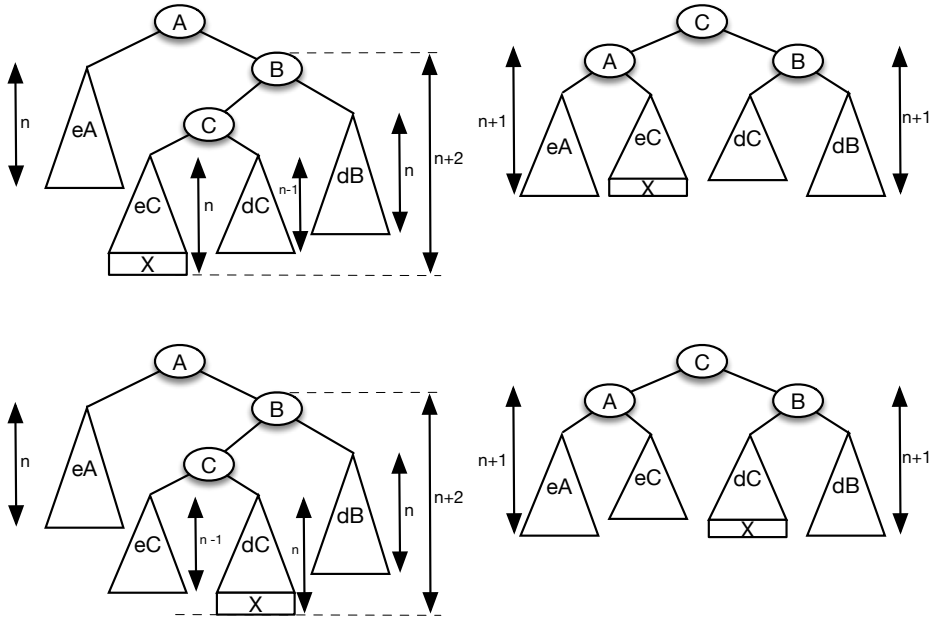


Para recalcularmos os factores de balanço dos nodos envolvidos temos que ter em atenção o factor de balanço da árvore onde foi inserido o novo elemento (na figura acima, o factor de balanço do nodo C).

O caso mais simples acontece quando o factor de balanço desse nodo é `BAL`. Corresponde ao caso em que foi esse o nodo criado para inserir o novo elemento.



Nos outros casos, apesar de o factor de balanço da raiz da árvore passar a `BAL`, o das suas subárvores não.

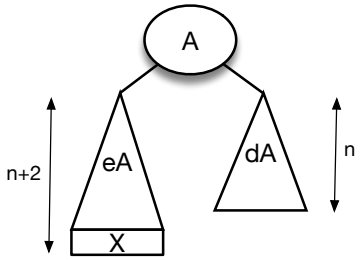


Analisados todos os casos estamos em condições de definir a função `fixRight` que faz os reajustes apresentados.

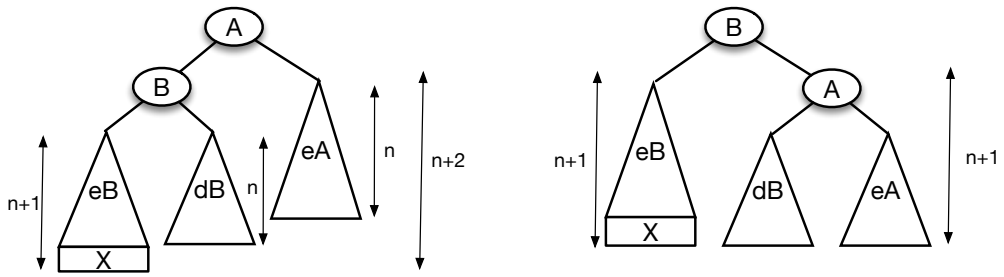
```
AVLTree fixRight (AVLTree a) {
  AVLTree b, c;
  b=a->right;
  if (b->bal == RIGHT){
     a->bal=b->bal=BAL;
     a=rotateLeft (a);
  } else {
     c=b->left;
     switch (c->bal) {
        case LEFT:  a->bal=BAL;
                    b->bal=RIGHT;
                    break;
        case RIGHT: a->bal=LEFT;
                    b->bal=BAL;
                    break;
        case BAL:   a->bal=b->bal=BAL;
     }
```

```
        c->bal = BAL;
        a->right = rotateRight (b);
        a=rotateLeft (a);
    }
    return a;
}
```

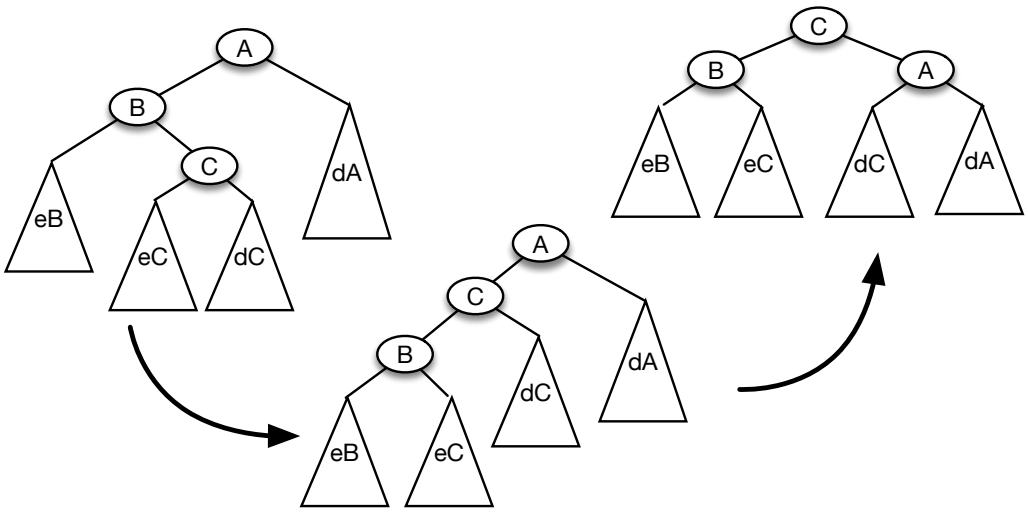
O caso que nos falta tratar é em tudo simétrico. Acontece quando a inserção se faz na sub-árvore esquerda de uma árvore com factor de balanço `LEFT` e essa inserção aumenta a altura da sub-árvore.



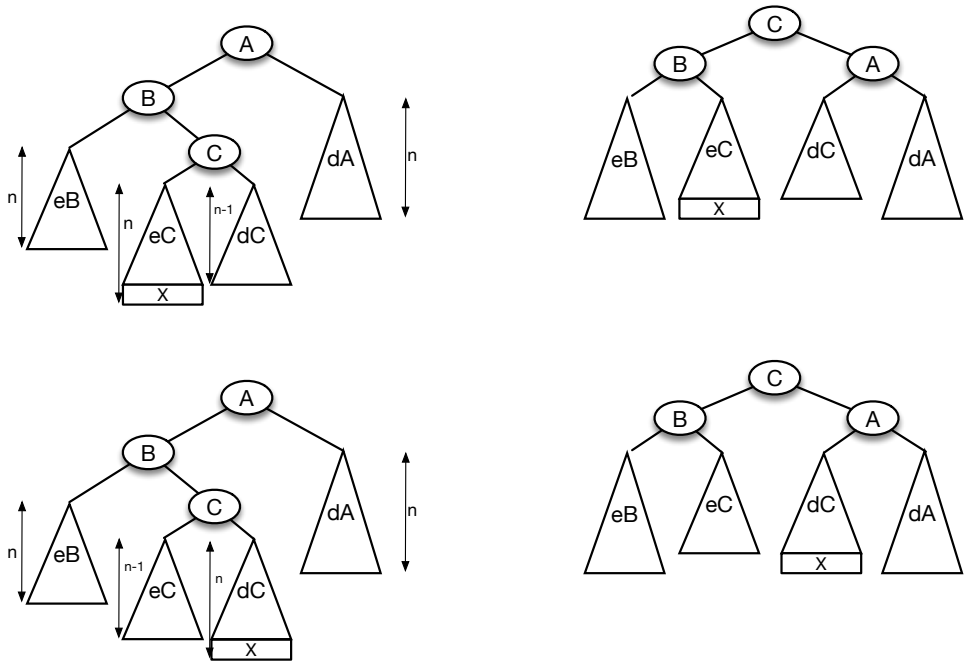
Mais uma vez o ajuste necessário vai depender do factor de balanço da árvore onde foi inserido o novo elemento.
No caso do factor de balanço ser também `LEFT`, uma rotação para a direita da árvore repõe o balanceamento.



No caso do factor de balanço dessa subárvore ser `RIGHT` teremos que fazer o reajuste em dois passos.



Também aqui, para actualizar correctamente os factores de balanço, teremos que inspeccionar o factor de balanço de mais um nodo da árvore (o nodo C na figura acima).



Esta simetria também se traduz no código que a implementa.

```
AVLTree fixLeft (AVLTree a) {
  AVLTree b, c;
  b=a->left;
  if (b->bal == LEFT){
     a->bal=b->bal=BAL;
     a=rotateRight (a);
  } else {
     c=b->right;
     switch (c->bal) {
        case LEFT:  a->bal=RIGHT;
                    b->bal=BAL;
                    break;
        case RIGHT: a->bal=BAL;
                    b->bal=LEFT;
                    break;
        case BAL:   a->bal=b->bal=BAL;
     }
     c->bal = BAL;
     a->left = rotateLeft (b);
     a=rotateRight (a);
  }
```

```
  return a;
}
```

Estamos finalmente em condições de apresentar a definição da função `updateAVLRec` referida na página 52.

```
AVLTree updateAVLRec (AVLTree a, int k, int i, int *g, int *u){

  if (a == NULL) {
     a = malloc (sizeof (struct avl));
     a->key=k; a->info=i; a->bal=BAL;
     a->left=a->right=NULL;
     *g=1; *u=0;
  } else if (a->key==k) {
     a->info=i;
     *g=0; *u=1;
  } else if (a->key > k) {
     a->left = updateAVLRec (a->left, k, i, g, u);
     if (*g == 1)
        switch (a->bal) {
            case LEFT: a=fixLeft (a); *g=0; break;
            case BAL:  a->bal=LEFT; break;
            case RIGHT: a->bal=BAL; *g=0; break;
        }
  } else {
    a->right = updateAVLRec (a->right, k, i, g, u);
     if (*g == 1)
        switch (a->bal) {
            case RIGHT: a=fixRight (a); *g=0; break;
            case BAL:  a->bal=RIGHT; break;
            case LEFT: a->bal=BAL; *g=0; break;
        }
  }
  return a;
}
```

A Exercícios de testes

1. *(Exame de recurso, 2007/2008)*
   Pretende-se usar uma tabela de *hash* para armazenar o conjunto das matrículas dos carros com acesso aos parques reservados da universidade. Assumindo que o tratamento das colisões é feito usando *chaining*, esta tabela pode ser implementada com o tipo ao lado.

```
#define SIZE 1009
typedef struct no {
   char matricula[6];
   struct no *next;
} No;
typedef No *Tabela[SIZE];
```

   - Implemente uma função `int hash(char matricula[6]);` de *hash* razoável para este problema.
   - Implemente a função `int insert(Tabela t, char matricula[6])` de inserção de uma matrícula na tabela, garantindo que não se armazenam matrículas repetidas.

2. *(Exame de recurso, 2007/2008)*
   Considere o seguinte tipo para implementar uma árvore AVL de inteiros.
   Implemente a função `Arvore rr(Arvore arv)` que faz uma rotação simples para a direita numa determinada sub-árvore. Não se esqueça de actualizar o factor de balanço nos nós envolvidos na rotação.

```
typedef struct no {
   int info;
   int bal;
   struct no *esq, *dir;
} No;
typedef No *Arvore;
```

3. *(Exame de recurso, 2007/2008)*
   Analise a complexidade da função `cols` que calcula os factores de balanço de uma árvore. Assuma que árvore está balanceada e que a função `altura` executa em tempo linear no tamanho da árvore de entrada.

```
void bals(Arvore a) {
   if (!a) return;
   a->bal = altura(a->dir) - altura(a->esq);
   bals(a->esq);
   bals(a->dir);
}
```

4. *(Teste, 2007/2008)*

Pretende-se implementar um dicionário de sinónimos usando uma *tabela de hash*. Esta tabela pode ser definida da seguinte forma, onde a cada palavra está associada uma lista ligada com os seus sinónimos.
Implemente a função `void sinonimos(Dic d, char *pal)` que, dada uma palavra, imprime todos os seus sinónimos.

```
typedef struct s {
   char *sin;
   struct s *next;
} Sin;

typedef struct p {
   char *pal;
   Sin *sins;
   struct p *next;
} Pal;

#define TAM ...
typedef Pal *Dic[TAM];

int hash(char *pal);
```

5. *(Teste, 2007/2008)*
Relembre o conceito de *min-heap*. Implemente uma função que converte uma *min-heap* representada num array para uma *min-heap* representada como uma árvore do tipo indicado.

```
#define TAM ...

typedef int Heap[TAM];

typedef struct nodo {
   int val;
   struct no *esq, *dir;
} Nodo, *Tree;
```

6. *(Exame de recurso, 2008/2009)*
Desenhe os estados sucessivos da árvore AVL de números inteiros resultante da inserção por esta ordem dos números 50, 20, 10, 60, 40, 30, e 70. Identifique claramente todas as rotações efectuadas.

7. *(Exame de recurso, 2008/2009)*
Escreva uma função `buildBST` que, dado um *array de inteiros ordenado por ordem crescente* `arr` com `n` elementos, constrói uma *árvore binária de procura* balanceada com todos os elementos do array.

```
typedef struct node {
  into info;
  struct node *esq, *dir;
} *Node;
Node buildBST(int arr[], int n);
```

8. *(Exame de recurso, 2008/2009)*
Pretende-se uma estrutura de dados para representar conjuntos de inteiros (entre 0 e 1000), que suporte as seguintes operações: inserção, teste de pertença e listagem ordenada no écran. Defina um tipo de dados apropriado e as funções referidas, garantindo que o teste de pertença executa em tempo constante e a listagem ordenada em tempo linear (no número de elementos do conjunto).

9. *(Teste, 2008/2009)*

Considere o tipo ao lado para representar uma *min-heap*
Defina uma função `muda (MinHeap h, int pos, int valor)` que altera o valor do elemento que está na posição `pos` para `valor`, fazendo as trocas necessárias para que se mantenham as propriedades da heap `h`.

```
#define MaxH ...
typedef struct mHeap {
    int tamanho;
    int heap [MaxH];
    } *MinHeap;
```

10. *(Teste, 2008/2009)*
Considere o tipo ao lado para implementar árvores binárias de procura.
Defina uma função `int procura (ABPInt a, int l, int u)` que, dada uma árvore binária de procura **balanceada** e dois inteiros (l e u), determina se existe algum elemento da árvore compreendido entre esses inteiros (i.e., pertencente ao intervalo $]l \ldots u[$).

```
typedef struct nodo *ABPInt;
struct nodo {
   int valor;
   ABPInt esq, dir;
   };
```

11. *(Teste, 2008/2009)*
Analise a complexidade da função apresentada na alínea anterior não se esquecendo de identificar o melhor e pior casos.

12. *(Teste, 2008/2009)*
Considere o tipo ao lado para representar uma *min-heap*
Defina uma função que calcule o maior elemento da heap sem a percorrer necessariamente toda.

```
#define MaxH ...
typedef struct mHeap {
    int tamanho;
    int heap [MaxH];
    } MinHeap;
```

13. *(Época especial, 2009/2010)*
Recorde o que estudou sobre árvores AVL. Implemente a função int rdir(AVL *tptr) que efectua uma rotação simples para a direita.

A função deverá retornar 0 em caso de sucesso, e -1 no caso de a rotação ser impossível devido à estrutura da árvore.
Não é necessário ajustar os factores de balanceamento.
Note que a função recebe a árvore passada por referência (struct avlnode **tptr).

```
typedef struct avlnode {
    int value,balance;
    struct avlnode *esq,dir;
} *AVL;
```

14. *(Teste, 2009/2010)*
Recorde o que estudou sobre árvores AVL.

Represente graficamente a evolução de uma árvore AVL quando é efectuada a seguinte sequência de inserções: 10, 20, 30, 70, 40, 50. Não se esqueça de indicar os factores de balanceamento de cada nó.

15. *(Teste, 2009/2010)*
Recorde o que estudou sobre Tabelas de Hash e responda, justificando, às seguintes perguntas:

    (a) Qual o pior caso do tempo de execução de uma pesquisa numa tabela de hash com n elementos inseridos, quando se utiliza *chaining* como mecanismo de resolução de colisões.

    (b) Qual o tempo de execução de percorrer todas as entradas, por ordem dos seus valores, numa tabela de hash com n elementos, quando se utiliza *chaining* como mecanismo de resolução de colisões.

    (c) Em que circunstâncias optaria pela utilização de uma Tabela de Hash? Relacione com as suas respostas às alíneas anteriores,

16. *(Época especial, 2010/2011)*
Considere uma tabela de Hash implementada sobre um array de tamanho 7 para armazenar números inteiros. A função de hash utilizada é $h(x) = x\%7$ (em que % representa o resto da divisão inteira). O mecanismo de resolução de colisões utilizado é *open addressing* com *linear probing*.

    - Apresente a evolução desta estrutura de dados quando são inseridos os valores 1, 15, 14, 3, 9, 5 e 27, por esta ordem.
    - Descreva o processo de remoção de um elemento ensta estrutura de dados, exemplificando com a remoção do valor 1 depois das inserções acima.

17. *(Exame de recurso, 2010/2011)*
Apresente a evolução de uma árvore AVL (inicialmente vazia) onde se inseriram as seguintes chaves (pela ordem apresentada): 3, 8, 2, 9, 5, 1, 4, 7, e 6.

18. *(Teste, 2010/2011)*
Suponha que está a utilizar uma tabela de Hash com resolução de colisões por Chaining, e em que o factor de carga atingiu os 50%.

    - Apresente uma definição de factor de carga e indique qual o valor máximo para este parâmetro no caso do método de resolução de colisões apresentado.
    - Qual o número de comparações no melhor caso de uma pesquisa falhada na tabela indicada? E no pior caso? Justifique a sua resposta e apresente o resultado da sua análise utilizando notação assimptótica.

19. *(Época especial, 2011/2012)*
Considere as seguintes definições para representar uma *min-heap* dinâmica de inteiros.
Defina uma função `int *ordenados (MinHeap h)` que calcula um array ordenado com os elementos da min-heap. Assuma que existe uma função `void bubbleDown (MinHeap h)`.
Note que a função a definir tem que alocar o espaço para o array resultado e que destrói o conteúdo da min-heap.

```
typedef struct minheap {
    int size;
    int used;
    int *values;
} *MinHeap;
```

20. *(Época especial, 2011/2012)*
Considere as seguintes definições para representar tabelas de *hash* dinâmicas com tratamento de colisões por *chaining*.
Diga o que entende por factor de carga de uma destas tabelas e defina uma função `float loadFactorC (HashTableChain t)` que calcula o factor de carga de uma tabela.

```
typedef struct entryC {
   char key[10];
   void *info;
   struct entryC *next;
} EntryChain;
typedef struct hashT {
   int hashsize;
   EntryChain *table;
} *HashTableChain;

int hash (int size, char key[]);
```

21. *(Época especial, 2011/2012)*
Na sequência do exercício anterior suponha que as chaves da tabela são palavras (i.e., sequências de letras terminada com o carater nulo `'\0'`), e que existe calculado em `int rank[26]` a significância de cada letra (quanto mais frequente é a letra, menor é o seu *rank*). Defina uma função de hash `int hash (int size, char[])` que tenha em consideração a significância de cada letra (bem como o tamanho da tabela).

22. *(Época especial, 2011/2012)*
Suponha que, numa árvore AVL inicialmente vazia, foram inseridas as chaves 10, 15, 8, 9 e 20 (por esta ordem). Note que nenhuma das inserções provocou o desbalanceamento da árvore. Apresente uma sequência de inserções balanceadas na dita árvore que façam com que a chave que fica na raíz da árvore passe a ser 20.

23. *(Exame de recurso, 2011/2012)*
Considere as seguintes definições para representar uma *min-heap* dinâmica de inteiros.

```
typedef struct minheap {
  int size;
  int used;
  int *values;
} *MinHeap;
```

   (a) Suponha que numa destas estruturas está guardada uma *min-heap* com tamanho (`size`) 100, com 10 elementos (i.e., `used` tem o valor 10) e que as 10 primeiras posições do vector `values` têm os valores `4, 10, 21, 45, 13, 25, 22, 60, 100, 20`.

   Diga qual o conteúdo desse vector após a remoção de dois elementos. Justifique a sua resposta desenhando as árvores correspondentes à *min-heap* antes e depois das referidas remoções.

(b) Considere a função ao lado que constrói uma árvore binária a partir de uma *min-heap*.
Faça a análise da complexidade da função `hToAux`, em função do tamanho da *min-heap*.

```
BTree heapToTree (MinHeap h) {
   return hToTAux (h, 0);
}
BTree hToTAux (MinHeap h, int r){
   BTree new;
   if (r >= h->used) return NULL;
   new = malloc (sizeof (struct btree));
   new->value = h->values[r];
   new->left = hToTAux (h, 2*r+1);
   new->right = hToTAux (h, 2*r+2);
   return new;
}
```

24. *(Exame de recurso, 2011/2012)*
Considere as seguintes definições para representar tabelas de *hash* dinâmicas com tratamento de colisões por *open addressing*.

```
#define STATUS_FREE    0
#define STATUS_USED    1
#define STATUS_DELETED 2

typedef struct entryO {
     int status;
     char *key;
     void *info;
} EntryOAdd;
```

```
typedef struct hashT {
    int size;
    int used;
    EntryOAdd *table;
} *HashTableOAddr;
```

Defina uma função `void doubleTable (HashTableOAddr h)` que duplica o tamanho de uma tabela (preservando a informação armazenada). Note que a tabela é passada por referência e que deve ser alterada.

25. *(Exame de recurso, 2011/2012)*
Pretende-se uma estrutura de dados para representar multi-conjuntos (i.e., conjuntos com elementos repetidos) de inteiros entre 0 e 1000, que suporte as seguintes operações: inserção, número de ocorrências de um elemento num multi-conjunto, e listagem ordenada do multiconjunto no écran.

Defina um tipo de dados apropriado e as funções referidas, garantindo que o cálculo do número de ocorrências executa em tempo constante e a listagem ordenada em tempo linear (no número de elementos do conjunto).

26. *(Teste, 2011/2012)*
Considere as seguintes definições para representar uma *min-heap* de inteiros.

(a) Suponha que numa destas estruturas está guardada uma *min-heap* com tamanho (`size`) 100, com 10 elementos (i.e., `used` tem o valor 10) e que as 10 primeiras posições do vector `values` têm os valores `4, 10, 21, 45, 13, 25, 22, 60, 100, 20`.

Diga qual o conteúdo desse vector após a inserção do número 6. Justifique a sua resposta desenhando as árvores correspondentes à *min-heap* antes e depois da referida inserção.

```
typedef struct minheap {
   int size;
   int used;
   int values[];
} *MinHeap;
```

(b) Defina uma função `int minHeapOK (MinHeap h)` que testa se a min-heap está correctamente construída (i.e., se todos os caminhos da raíz até uma folha são sequências crescentes). Certifique-se que a solução que apresentou tem um custo linear no tamanho da input.

27. *(Teste, 2011/2012)*
Considere as seguintes definições para representar tabelas de *hash* dinâmicas com tratamento de colisões por *chaining*.

```
typedef struct entry {
   char key[10];
   void *info;
   struct entry *next;
} *Entry;
```

```
typedef struct hashT {
   int hashsize;
   Entry *table;
} *HashTable;
```

```
int hash(int hashsize, int key[]);
```

Defina uma função `HashTable newTable(int hashsize)` que inicializa uma tabela de tamanho `hashsize`. Note que deve ser alocada a memória necessária e que todas as entradas da tabela devem ser inicializadas com a lista vazia.

28. *(Teste, 2012/2013)*
Considere uma tabela de *hash* (para implementar um conjunto de inteiros) com tratamento de colisões por *open addressing* (com *linear probing*) e em que a remoção de chaves é feita usando uma marca (de apagado).

Suponha que existem definidas as funções `add (int k)`, `remove (int k)`, e `exists (int k)`, sendo que esta última devolve verdadeiro/falso.

Suponha ainda que o tamanho da tabela é 7 e que a função de *hash* é `hash(x) = x % 7`.

Apresente a evolução da tabela quando, a partir de uma tabela inicialmente vazia, se executa a seguinte sequência de operações:

```
add 15; add 25; add 9; add 1; rem 9; add 38; rem 15; add 6; add 10;
```

29. *(Teste, 2012/2013)*
Considere as definições ao lado para implementar árvores AVL de inteiros. Defina uma função `int altura (AVL a)` que calcula a altura de uma AVL em tempo logarítmico no tamanho (número de nodos) da árvore.

```
#define Bal 0  // Balanceada
#define Esq -1 // Esq mais pesada
#define Dir 1  // Dir mais pesada

typedef struct avlNode *AVL;

struct avlNode {
    int bal; // Bal/Esq/Dir
    int valor;
    struct avlNode *esq,*dir;
}
```

30. *(Exame de recurso, 2012/2013)*
Uma definição alternativa de árvores AVL consiste em guardar, para cada nodo da árvore, a altura da árvore que aí se inicia em vez de guardar o factor de balanço desse nodo.

Considere então a seguinte definição de uma dessas árvores (de inteiros). Defina uma função `AVL rotateLeft (AVL a)` que faz uma rotação (simples) à esquerda na raíz de uma destas árvores. Assuma que a rotação é possível e não se esqueça de actualizar o campo `altura` dos nodos envolvidos.

```
typedef struct nodo{
      int valor;
      int altura;
      struct nodo *esq, *dir;
    } Node, *AVL;
```

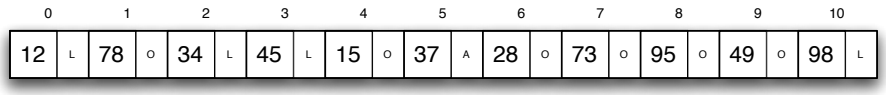
31. *(Exame de recurso, 2012/2013)*
Considere as definições ao lado para implementar tabelas de Hash dinâmicas com tratamento de colisões por open addressing e linear probing.
Note a existência de uma função `int hash (Key, int)` que recebe como argumentos uma chave e o tamanho da tabela.
Defina uma função `void remApagados (THash h)` que remove os elementos apagados de uma tabela, i.e., que efectua as operações necessárias para que deixem de existir células marcadas como apagadas.

```
#define Livre 0
#define Ocupado 1
#define Apagado 2

typedef struct key *Key;
struct celula {
  Key k;
  void *info;
 int estado; //Livre/Ocupado/Apagado
 }

typedef shash {
  int tamanho, ocupados, apagados;
  struct celula *Tabela;
} *THash;

int hash (Key, int);
```

32. *(Época especial, 2012/2013)*

Considere uma tabela de *Hash*, com *open addressing* e tratamento de colisões por *linear probing*. Considere ainda que, para implementar remoções, cada célula da tabela tem uma *flag* que pode tomar três valores possíveis: `L`, `O` ou `A` (Livre, Ocupada ou Apagada).

Assumindo que as chaves são inteiros e que a função de *hash* usada é `hash(n) = n%N`, considere o seguinte estado da tabela (para `N=11`).



Partindo do princípio que a função `procura`, recebe o valor de uma chave e retorna a posição da tabela onde essa chave se encontra (e `-1` caso a chave não se encontre na tabela), qual o resultado de fazer, no estado apresentado, `procura(12)` e `procura(37)`. Para cada um dos casos, indique quais as posições da tabela que são consultadas.

33. *(Época especial, 2012/2013)*
Considere a definição ao lado para implementar uma árvore AVL de inteiros.
Defina uma função que calcula a altura de uma árvore em tempo logarítmico em relação ao número de nodos da árvore. Justifique brevemente que a sua função tem de facto complexidade logarítmica.

```
#define Bal 0
#define Esq 1
#define Dir (-1)
typedef struct avlnode {
  int valor;
  int balanco;
  struct avlnode *esq, *dir;
} *AVL;
```

34. *(Teste, 2013/2014)*
A função `heap2ALV` apresentada ao lado transfere um conjunto de números inteiros armazenado numa *minheap* para uma árvore AVL. Considere que a função `extractMin` remove um elemento da *minheap* e retorna zero se a *heap* é vazia. Considere ainda que a função `insertTree` faz a inserção balanceada numa árvore AVL.

```
Node* heap2AVL (Heap *h) {
  int x;
  Node *t = NULL;
  while (extractMin(h, &x))
    t = insertTree (t, x);
  return t;
}
```

(a) Considere que a função é invocada com a *minheap* $[10, 20, 30, 40, 60, 50, 70]$. Apresente visualmente o conteúdo da *heap* `h` e da árvore AVL `t` em todas as iterações do ciclo.

(b) Analise a complexidade assimptótica desta função em função do tamanho (número de elementos) da heap argumento.

35. *(Teste, 2014/2015)*
Considere o seguinte tipo de dados para armazenar um conjunto de pares – strings com um máximo de 10 caracteres – e inteiros, utilizando uma árvore AVL ordenada pela string.

```
typedef struct data {
     int dados;
} Data;

typedef struct node {
     int balance; // -1 left higher,
                  //  0 balanced,
                  //  1 right higher
     char key[11];
     Data info;
     struct node *left, *right;
} Node;

typedef Node *Dictionary;
```

(a) Apresente a evolução do estado da árvore AVL à medida que são inseridos (numa árvore inicialmente vazia) os seguintes pares: (JOSE, 1), (MANUEL, 1), (JOAQUIM, 1), (MANUEL, 2), (ANTONIO, 1), (MANUEL, 3), (JOSE, 2), (ANDRE, 1), (PEDRO, 1), (PAULO, 1). Considere que, em caso de inserções repetidas, uma nova cópia da chave é inserida na sub-árvore esquerda.

(b) *(Teste, 2014/2015)*
Implemente a seguinte função que retorna o número de cópias encontradas de uma dada string. Certifique-se que a função definida não tem que percorrer toda a árvore.

```
int allCopies(Dictionary dic, char key[11]);
```

Efectue a análise assimptótica do tempo de execução da função `allCopies`.

36. *(Exame de recurso, 2014/2015)*
Considere o tipo definido à direita para representar uma árvore binária de inteiros.
Uma heap é uma árvore binária em que a raiz é menor ou igual a todos os outros nodos e as subárvores são ainda heaps.

```
typedef struct node {
    int value;
    struct node *left, *right;
} Node, *BTree;
```

Defina uma função `int heapOK (BTree a)` que testa se uma dada árvore é uma heap (retornando `1` em caso afirmativo e `0` caso não seja).

Note que se pretende apenas testar a relação entre os valores armazenados (e não a forma da árvore).

Assegure-se que a função que definiu executa, no pior caso em tempo linear no número de elementos da árvore, e no melhor caso em tempo constante.

37. *(Época especial, 2014/2015)*
Relembre a definição de *min-heap*. O tipo de dados ao lado permite a definição de heaps dinâmicas de inteiros (campos `size` e `used` são usados para guardar o tamanho do array `heap` e o número de elementos efectivamente armazenados).

```
typedef struct {
 int   size;
 int   used;
 int  *heap;
} Heap;
```

Nas operações de inserção/remoção na heap consideram-se os seguintes requisitos adicionais:

- se a heap estiver cheia, o tamanho do array é duplicado para obter espaço para a inserção;
- se, removendo o mínimo da heap, o array tiver tamanho superior a 2 e ficar com pelo menos 75% do espaço livre, o seu tamanho é reduzido para metade.

Apresente uma implementação da operação `void insertHeap(Heap *h, int x)`.

38. *(Teste, 2015/2016)*
Relembre a definição de uma *min-heap* armazenada num array e a função `void bubbledown (int heap[], int N, int i)` que recebe um array `heap` com `N` elementos e faz o *bubble-down* do elemento que está no índice `i`.
Considere agora a função que transforma um array arbitrário numa *min-heap*.

```
void heapify (int v[], int N){
  int i;
  for (i=(N-2)/2; i>=0; i--)
    bubbledown (v,N,i);
}
```

(a) Diga qual o resultado da função `heapify` quando invocada com um array `v` com os seguintes 7 elementos por esta ordem: {`50, 20, 42, 13, 2, 12, 36`}. Justifique a sua resposta apresentando o estado do array no final de cada iteração do ciclo.

(b) Note que a função `bubbledown` só é aplicada a metade dos elementos do array. Além disso,

- para 1/4 dos elementos (a metade superior dessa metade) a função `bubbledown` faz no máximo 1 iteração.
- para 1/8 dos elementos a função `bubbledown` faz no máximo 2 iterações, e assim sucessivamente.

Com base nestas observações, analise a complexidade da função `heapify` apresentada.

39. *(Exame de recurso, 2015/2016)*

Considere a seguinte definição de um tipo para representar árvores AVL de inteiros (**balanceadas**). Considere ainda a definição da função `deepest` que determina o nodo mais profundo de uma árvore binária, retornando o nível em que ele se encontra.

(a) Mostre que a função apresentada tem uma complexidade linear no número de elementos da árvore argumento.

(b) Apresente uma definição alternativa, substancialmente mais eficiente, tirando partido da informação presente no factor de balanço de cada nodo. Diga qual a complexidade da função que definiu.

```
typedef struct avlnode {
  int value;
  int bal; // Left/Bal/Right
  struct avlnode *left, *right;
} *AVLTree;

int deepest (AVLTree *a) {
  AVLTree l, r;
  int hl, hr, h;
  if (*a == NULL) h = 0;
  else {
    l = (*a)->left; r = (*a)->right;
    hl = deepest (&l); hr = deepest (&r);
    if ((hl>0) && (hl > hr)) {
      *a = l; h = hl+1;}
    else if (hr>0) {
      *a = r; h = hr+1;}
    else h=1;
  }
  return r;
}
```

40. *(Exame de recurso, 2015/2016)*
Considere as seguintes definições para implementar tabelas de hash de números inteiros (em open-address e com chaining).

(a) Defina uma função `int fromChain (HashChain h1, HashOpen h2)` que preenche a tabela `h2` com as chaves presentes na tabela `h1`. Note que ambas as tabelas têm a mesma dimensão.

(b) Que alterações deveria efectuar na definição anterior se as tabelas tivessem dimensão diferente?

```
#define HSIZE 1000
int hash (int chave, int size);
typedef struct lista {
  int valor;
  struct lista *prox;
} *HashChain[HSIZE];
typedef struct celula {
  int estado; // 0 - Livre
              // 1 - Ocupado
              // 2 - Removido
  int valor;
} HashOpen[HSIZE];
```

41. *(Teste, 2016/2017)*

Na implementação de tabelas de hashing com *open addressing* muitas vezes guarda-se para cada chave inserida o número de colisões que essa inserção teve que resolver (`probe == 0` significa que não houve qualquer colisão).
Considere a definição ao lado utilizada para implementar essas tabelas e apresente o resultado de inserir as chaves `40, 80, 60, 260, 54, 65, 140` por esta ordem, numa tabela inicialmente vazia usando linear probing.
Assuma que a tabela tem tamanho `Hsize=13` e que a função de hash é apenas `hash (t) = t % Hsize`.

```
#define Hsize ...
#define FREE    0
#define USED    1
#define DELETED 2
typedef struct entry {
  int key;
  int probe;
  int status;
} Entry;
typedef Entry THash [Hsize];
```

42. *(Teste, 2016/2017)*
Relembre a definição de árvores AVL (de inteiros) apresentado ao lado e o algoritmo de inserção balanceada estudado.

    (a) Defina uma função `AVL maisProfundo (AVL a)` que, dada uma árvore AVL, determina um nodo da árvore que se encontra à profundidade máxima. A função retorna `NULL` no caso da árvore ser vazia e deverá executar em tempo logoraítmico no número de nodos da árvore.

    (b) Qual o resultado da função que definiu quando for aplicada à árvore que resulta de se inserirem, numa árvore inicialmente vazia, os valores `20, 40, 10, 50, 30, 15, 29` (por esta ordem)? Justifique.

```
#define LEFT   1
#define BAL    0
#define RIGHT -1
typedef struct avl {
  int value;
  int bal;
  struct avl *left, *right;
} *AVL;
```

43. *(Exame de recurso, 2016/2017)*

Considere a definição de tipos à direita para representar tabelas de Hash de inteiros usando open addressing com vectores dinâmicos e tratamento de colisões por linear probing. Considere ainda que

- a função de hash usada é `hash(x) = x%tamanho`
- sempre que o número de células livres é menor do que 25% do tamanho da tabela esta é realocada para uma tabela de tamanho `2*tamanho+1`.
- sempre que o número de células apagadas é maior ou igual ao número de células ocupadas é feita uma *garbage collection*.

```
struct celula {
  int k;
  char estado; //L/O/A
}
typedef shash {
  int tamanho,
      ocupados,
      apagados;
  struct celula *Tabela;
} *THash;
```

Apresente a evolução de uma tabela, inicialmente vazia e de tamanho (inicial) `7`, quando são efectuadas as seguintes operações de inserção (`ins`) e remoção (`del`) (por esta ordem).

`ins 10`, `ins 4`, `ins 16`, `del 10`, `del 4`, `ins 9`, `ins 13`, `ins 0` `ins 7`

44. *(Exame de recurso, 2016/2017)*
Considere as definições (apresentadas à direita) de listas ligadas e de árvores AVL em que cada nodo possui ainda uma marca para assinalar nodos apagados.
Defina uma função `AVL fromList (LInt l, int n)` que constrói uma árvore AVL a partir de uma lista **ordenada** com `n` elementos.
Não se esqueça de garantir que (1) a árvore produzida está balanceada, (2) os factores de balanço estão correctamente calculados e (3) a função executa em tempo linear no número de elementos da lista.

```
typedef struct llint {
  int value;
  struct llint *next;
} *LInt;

typedef struct avl {
  int value;
  int bal, deleted;
  struct avl *left, *right;
} *AVL;
```

45. *(Época especial, 2016/2017)*
Considere uma tabela de hash de inteiros implementada num array dinâmico com open-addressing e linear probing, com tamanho inicial `size=1` e função de hash `hash(x,size)=x%size` inicialmente vazia. Considere ainda que

- sempre que o factor de carga da tabela excede 50% a tabela é redimensionada para `2*size+1`.
- sempre que o factor de carga da tabela baixa de 33% a tabela é redimensionada para `2*size/3`.

Apresente o estado da tabela após as seguintes operações:

- inserção dos números 1, 3, 5, 8, 10 e 15.
- remoção dos números 5 e 15

46. *(Época especial, 2016/2017)*
Considere a seguinte definição de um tipo para armazenar tabelas de hash de inteiros tal como descritas na alínea anterior.
Defina a função de remoção de um elemento tal como referido atrás.

```
#define LIVRE   0
#define APAGADO 1
#define OCUPADO 2
struct entry {
    int value, status;
};
typedef struct thash {
  int ocupados, tamanho;
  struct entry *tabela;
} *THash;
```

47. *(Época especial, 2016/2017)*
Relembre a definição de árvores binárias (de inteiros) e a função de rotação (à direita) usada na inserção AVL.
Considere ainda a definição ao lado da função `spine` que transforma uma árvore binária numa espinha (i.e., uma árvore em que todas as sub-árvores esquerdas são nulas).

- Defina a função `rotateRight` usada nessa definição.
- Escreva uma relação de recorrência que traduza o tempo de execução da função `spine` no caso da árvore recebida como argumento já ser uma *espinha* e apresente uma solução dessa recorrência.

```
typedef struct nodo {
   int valor;
   struct nodo *esq, *dir;
} *ABin;

AVL rotateRight (AVL a);

AVL spine (AVL a){
   if (a!=NULL) {
      while (a->esq != NULL)
         a=rotateRight (a);
      a->dir = spine(a->dir);
   }
   return (a);
}
```