

Representação de grafos

```
// assume listas de adjacência ordenadas
// por ordem crescente do vértice destino
int inDegree_sol (GraphL g, int j, int n) {
    int i, k = 0;
    struct edge *p;

    for (i=0; i<n; i++) {
        for (p = g[i]; p && p->dest < j ; p=p->next) ;
        if (p->dest == j) k++;
    }

    return k;
}
```

```
int outDegree_sol (GraphL g, int i) {
    int n = 0;
    struct edge *p;

    for (p = g[i]; p ; p=p->next) n++;

    return n;
}
```

```
// assume listas de adjacência ordenadas
// por ordem crescente do vértice destino
void graphLtoM_sol(GraphM gm, GraphL gl, int n) {
    int i, j;
    struct edge *p;

    for (i = 0; i<n; i++)
        for (j = 0, p = gl[i]; j<n; j++) {
            if (p && p->dest == j) {
                gm[i][j] = p->weight;
                p = p -> next;
            }
            else
                gm[i][j] = NE;
        }
}

void graphMtoL_sol(GraphL gl, GraphM gm, int n) {
    int i, j;
    struct edge *head, *new;

    for (i = 0; i<n; i++) {
        head = NULL;
        for (j = n-1; j>=0; j--)
            if (gm[i][j] != NE) {
                new = malloc(sizeof(struct edge));
                new -> dest = j;
                new -> weight = gm[i][j];
                new -> next = head;
                head = new;
            }
        gl[i] = head;
    }
}
```

Depth first → travessia em profundidade

```
int color[MAX];
```

```
void df_visit(GraphL g, int s) {
    struct edge *p;

    color[s] = GRAY;
    printf("%d GRAY\n", s);

    for (p=g[s]; p; p=p->next) {
        if (color[p->dest] == WHITE)
            df_visit (g, p->dest);
    }
}

color[s] = BLACK;
printf("%d BLACK\n", s);
}
```

```
void dfs(GraphL g, int n) {
    int u;

    for (u=0; u<n; u++)
        color[u] = WHITE;

    for (u=0; u<n; u++)
        if (color[u] == WHITE) df_visit (g, u);

}
```

Breadth first → travessia em largura

```
void bf_visit(GraphL g, int s) {  
    struct edge *p;  
    int q[MAX], first=0, last=0, u, v;  
  
    color[s] = GRAY;  
    printf("%d GRAY\n", s);  
    q[last++] = s;  
  
    while (first < last) {  
        u = q[first++];  
        for (p=g[u]; p; p=p->next) {  
            v = p->dest;  
            if (color[v] == WHITE) {  
                color[v] = GRAY;  
                printf("%d GRAY\n", v);  
                q[last++] = v;  
            }  
        }  
        color[u] = BLACK;  
        printf("%d BLACK\n", u);  
    }  
}
```

```
void bfs(GraphL g, int n) {  
    int u;  
  
    for (u=0; u<n; u++)  
        color[u] = WHITE;  
  
    for (u=0; u<n; u++)  
        if (color[u] == WHITE) bf_visit (g, u);  
}
```

caminhos mais curtos e as distâncias

```
// constrói árvore  
// não calcula distâncias  
  
int shortestPath_sol (GraphL g, int s, int d, int path[]) {  
    struct edge *p;  
    int q[MAX], parent[MAX], first=0, last=0, i, u, v, c=0;  
    char vis[MAX];  
  
    for (u=0; u<MAX; u++) {  
        parent[u] = -1; iniciar  
        vis[u] = 0;  
    }  
  
    vis[s] = 1;  
    q[last++] = s;  
  
    while (first < last) {  
        u = q[first++];  
        for (p=g[u]; p; p=p->next) {  
            v = p->dest;  
            if (!vis[v]) {  
                vis[v] = 1;  
                parent[v] = u;  
                q[last++] = v;  
            }  
        }  
    }  
  
    if (parent[d]==-1) return -1;  
  
    // calcular distância de s a d  
    u = d;  
    while (u != -1) {  
        u = parent[u];  
        c++;  
    }  
  
    u = d;  
    i = c-1;  
    while (i>=0) {  
        path[i] = u;  
        u = parent[u];  
        i--;  
    }  
  
    return c;  
}
```

// não constrói árvore

```
void dists_sol (GraphL g, int s, int dist[]) {  
    struct edge *p;  
    int q[MAX], first=0, last=0, u, v;  
    char vis[MAX]; → em vez de color  
  
    for (u=0; u<MAX; u++) { iniciar dist.  
        dist[u] = -1;  
        vis[u] = 0;  
    }  
  
    vis[s] = 1;  
    dist[s] = 0;  
    q[last++] = s;  
  
    while (first < last) {  
        u = q[first++];  
        for (p=g[u]; p; p=p->next) {  
            v = p->dest;  
            if (!vis[v]) {  
                vis[v] = 1;  
                dist[v] = dist[u]+1;  
                q[last++] = v;  
            }  
        }  
    }  
}
```

```
// constrói árvore  
// calcula todas as distâncias (poderia não o fazer)  
  
int maisLonga_sol (GraphL g, int s, int path[]) {  
    struct edge *p;  
    int q[MAX], parent[MAX], dist[MAX], first=0, last=0, i, u, v, c=0;  
    char vis[MAX];  
  
    for (u=0; u<MAX; u++) {  
        dist[u] = -1;  
        vis[u] = 0;  
        parent[u] = -1;  
    }  
  
    vis[s] = 1;  
    dist[s] = 0;  
    q[last++] = s;  
  
    while (first < last) {  
        u = q[first++];  
        for (p=g[u]; p; p=p->next) {  
            v = p->dest;  
            if (!vis[v]) {  
                vis[v] = 1;  
                dist[v] = dist[u]+1;  
                parent[v] = u;  
                q[last++] = v;  
            }  
        }  
    }  
  
    c = dist[u]+1;  
    i = c-1;  
    while (i>=0) {  
        path[i] = u;  
        u = parent[u];  
        i--;  
    }  
  
    return c;  
}
```

Algoritmo de Prim → Árvore geradora de custo mínimo

```

int PrimMST (Graph g, int cost[], int ant[]){
    int res=0; int v;
    EList it;
    int color[NV];
    struct fringe ff, *f;
    int fringesize;
    f = &ff;
    for (v=0; v<NV; color[v++] = 0)
    ;
    initFringe(f); fringesize=0;
    v=0;
    color[v]=1; cost[v]=0;
    addEdgeFringe(f, v, cost); fringesize++;
    while (fringesize >0) {
        v = getEdge (f, cost); fringesize--;
        res+=cost[v];
        color[v]=2; //BLACK
        for (it=g[v]; it; it=it->next)
            if ((color[it->dest] == 0) || // WHITE
                (color[it->dest] == 1 && // GREY
                 cost[it->dest] > it->cost)) {
                ant[it->dest]=v;
                cost[it->dest]=it->cost;
                if (color[it->dest] == 0) { // white
                    addEdgeFringe(f, it->dest, cost);
                    fringesize++;
                    } else updateFringe(f, it->dest, cost);
            }
        }
    return res;
}

```

Algoritmo de Dijkstra → caminhos mais curtos

```

int dijkstraSP (Graph g, int v, int cost[], int ant[]){
    int res=0;
    int newcost;
    EList it;
    int color[NV];
    struct fringe ff, *f;
    int fringesize;
    f = &ff;
    for (v=0; v<NV; color[v++] = 0)
    ;
    initFringe(f); fringesize=0;
    color[v]=1; cost[v]=0;
    addEdgeFringe(f, v, cost); fringesize++;
    while (fringesize >0) {
        v = getEdge (f, cost); fringesize--;
        res++;
        color[v]=2; //BLACK
        for (it=g[v]; it; it=it->next)
            newcost = cost[v] + it->cost;
            if ((color[it->dest] == 0) // WHITE
                || ((color[it->dest] == 1) && // GREY
                     cost[it->dest] > newcost)) {
                ant[it->dest]=v;
                cost[it->dest]=newcost;
                if (color[it->dest] == 0) { // WHITE
                    addEdgeFringe(f, it->dest, cost);
                    fringesize++;
                    } else updateFringe(f, it->dest, cost);
            }
        }
    return res;
}

```

```

struct fringe {
    int size;
    int edges[NV];
};

```

A operação de inicialização da orla é feita em tempo constante.

```

int initFringe (Fringe f){
    f->size=0;
    return 0;
}

```

O mesmo acontecendo com a operação de adição de um novo vértice à orla.

```

int addEdgeFringe (Fringe f, int v, int cost[]){
    f->edges[f->size++]=v;
    return 0;
}

```

A operação de actualização do peso de um vértice da orla não precisa de fazer nenhuma acção uma vez que não existe qualquer ordem na forma como os vértices estão armazenados.

```

int updateFringe (Fringe f, int v, int cost[]){
    return 0;
}

```

A única operação que não é feita em tempo constante é a operação de selecção (e remoção) do vértice da orla com menor custo.

```

int getEdge (Fringe f, int cost[]){
    int i, minind, r;

    minind=0;
    for (i=1; i<f->size; i++)
        if (cost[f->edges[i]] < cost[f->edges[minind]])
            minind=i;
    r=f->edges[minind];
    f->size--;
    f->edges[minind]=f->edges[f->size];
    return r;
}

```

Esta função executa em tempo linear ao tamanho da orla. E esse valor está limitado pelo número de vértices do grafo.

Fecho transitivo

```
void MarshallTC_sol (Graph A, Graph R, int n) {
    int i, j, k;

    for (i=0 ; i<n ; i++)
        for (j=0 ; j<n ; j++)
            R[i][j] = A[i][j];

    for (k=0 ; k<n; k++)
        for (i=0 ; i<n ; i++)
            for (j=0 ; j<n ; j++)
                if (R[i][k] && R[k][j])
                    R[i][j] = 1;
}

int min (int x, int y) {
    return (x<=y) ? x : y;
}

void Floyd_Warshall_Distances_sol (Graph A, Graph D, int n) {
    int i, j, k;

    for (i=0 ; i<n ; i++)
        for (j=0 ; j<n ; j++)
            D[i][j] = A[i][j];

    for (k=0 ; k<n; k++)
        for (i=0 ; i<n ; i++)
            for (j=0 ; j<n ; j++)
                D[i][j] = min (D[i][j], D[i][k] + D[k][j]);
}
```

Componentes

```
void df_comp_sol(GraphL g, int s, int comp[], int c) {
    struct edge *p;

    comp[s] = c;

    for (p=g[s]; p; p=p->next) {
        if (comp[p->dest] == 0)
            df_comp_sol (g, p->dest, comp, c);
    }
}

void componentes_sol (GraphL g, int n, int comp[])
{
    int u, c=1;

    for (u=0; u<n; u++)
        comp[u] = 0;

    for (u=0; u<n; u++)
        if (comp[u] == 0) df_comp_sol (g, u, comp, c++);
}
```

File com prioridades
&
Heaps

```

#include <stdlib.h>           // 3 versões
#include "minheap.h"

void swap (Elem h[], int a, int b) {
    int t = h[a];
    h[a] = h[b];
    h[b] = t;
}

void initHeap_sol (Heap *h, int size) {
    h->values = calloc(size, sizeof(Elem));
    h->size = size;
    h->used = 0;
}
/* versão recursiva
void bubbleUp_sol (Elem *a, int i) {
    if (i!=0) {
        if (a[i] < a[PARENT(i)]) {
            swap(a, i, PARENT(i));
            bubbleUp_sol (a, PARENT(i));
        }
    }
}

// versão menos eficiente, melhor caso logarítmico!!
// pode-se acrescentar "else break" para recuperar o melhor caso constante
/*void bubbleUp_sol (Elem *a, int i) {
    int p;
    while (i!=0) {
        p = PARENT(i)
        if (a[i] < a[p])
            swap(a, i, p);
        i = p;
    }
}

void bubbleUp_sol (Elem *a, int i) {
    int p = PARENT(i);

    while (i!=0 && a[i] < a[p]) {
        swap(a, i, p);
        i = p;
    }
}

int insertHeap_sol (Heap *h, Elem x) {
    if (h->used == h->size) {
        h->values = realloc(h->values, 2*(h->size)*sizeof(Elem));
        h->size *= 2;
    }
    h->values[h->used] = x;
    (h->used)++;
    bubbleUp_sol(h->values, h->used-1);
    return 1;
}

#define PARENT(i) (i-1)/2 // os indices do array
#define LEFT(i) 2*i + 1
#define RIGHT(i) 2*i + 2

typedef int Elem; // elementos da heap.

typedef struct {
    int size;
    int used;
    Elem *values;
} Heap;

```

```

void bubbleDown_sol (Elem a[], int N) {
    int i = 0, min;

    while (LEFT(i) < N) {
        min = a[i] < a[LEFT(i)] ? i : LEFT(i);
        if (RIGHT(i) < N)
            min = a[min] < a[RIGHT(i)] ? min : RIGHT(i);
        if (min != i) {
            swap(a, i, min);
            i = min;
        } else break;
    }
}

void bubbleDown_sol_2 (Elem a[], int N) {
    int i = 0, min;

    while (LEFT(i) < N) {
        min = LEFT(i);
        if (RIGHT(i) < N && a[RIGHT(i)] < a[LEFT(i)])
            min = RIGHT(i);
        if (a[min] < a[i]) {
            swap(a, i, min);
            i = min;
        } else break;
    }
}

void bubbleDown_sol_3 (Elem a[], int N) {
    int i = 0, min;

    while (RIGHT(i) < N &&
           a[min = a[LEFT(i)] < a[RIGHT(i)] ? LEFT(i) : RIGHT(i)] < a[i]) {
        swap(a, i, min);
        i = min;
    }
    if (LEFT(i) < N && a[LEFT(i)] < a[i])
        swap(a, i, LEFT(i));
}

int extractMin_sol (Heap *h, Elem *x) {
    if (h->used > 0) {
        *x = h->values[0];
        h->values[0] = h->values[h->used-1];
        (h->used)--;
        bubbleDown_sol(h->values, h->used);
        return 1;
    } else return 0;
}

```

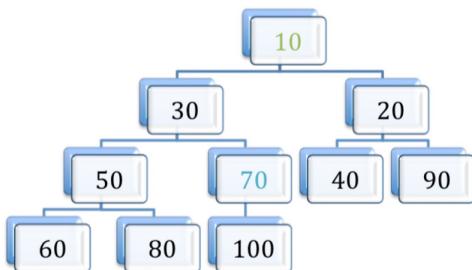
Uma heap é uma árvore binária, caracterizada por duas propriedades (invariantes de tipo):

- **Invariante de ordem:**

O valor associado a cada nó é *inferior ou igual* aos valores de todos os seus descendentes

- **Invariante de forma:**

- A árvore binária é *completa* (apenas o último nível pode não estar totalmente preenchido), e
- último nível é preenchido da esquerda para a direita, *sem "lacunas"*



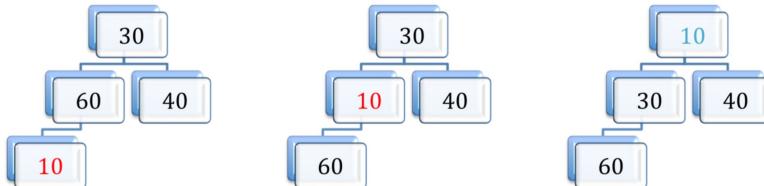
pode ser implementada ao nível físico pelo seguinte vector:

i	0	1	2	3	4	5	6	7	8	9
$v[i]$	10	30	20	50	70	40	90	60	80	100
Nível	1	2	2	3	3	3	3	4	4	4

filhos de $v[i] = v[i \times 2 + 1] \text{ e } v[i \times 2 + 2]$
 pai de $v[i] = v[(i-1)/2]$

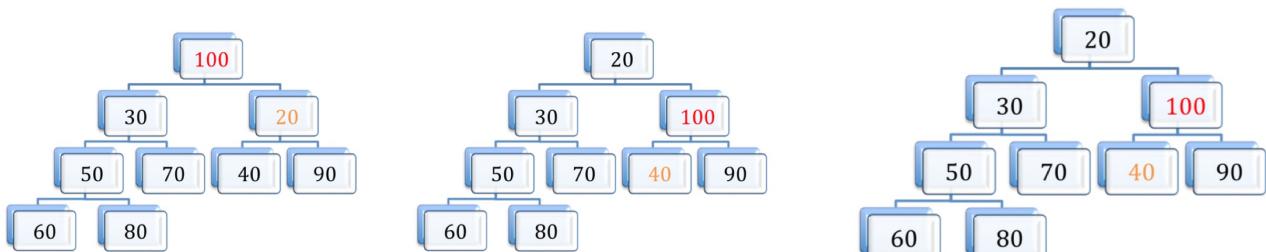
Algoritmo de Inserção:

1. Insere-se o novo elemento na primeira posição livre da heap, i.e. na posição mais à esquerda do último nível da heap;
2. Faz-se uma operação de **bubble-up**:
 Enquanto o elemento inserido for de valor inferior o seu pai na árvore, troca-se sucessivamente (ao longo de um caminho ascendente da heap) estes dois elementos.



Algoritmo de Extracção (operação *pull*):

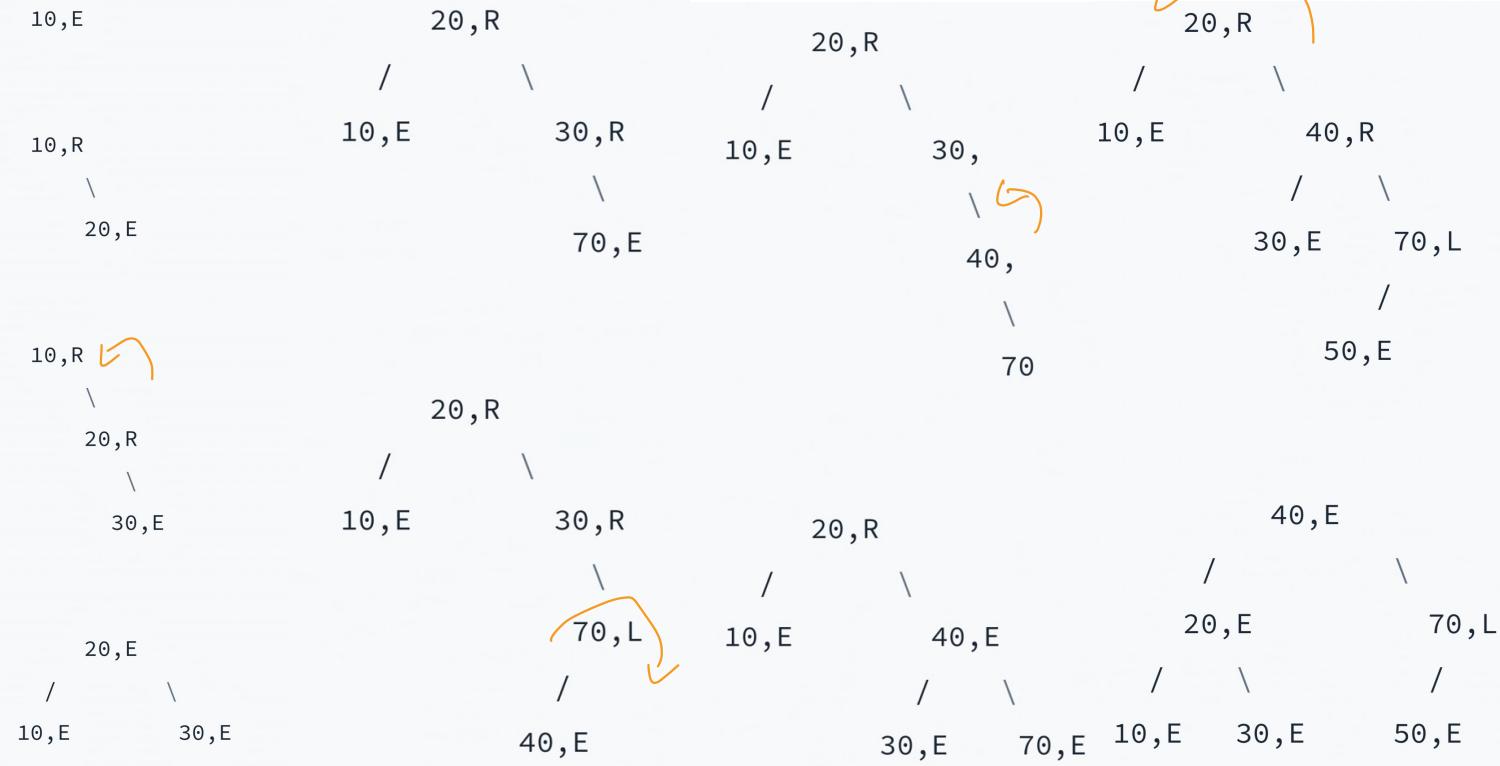
1. Remove-se o elemento inserido na última posição da heap, i.e. na posição mais à direita do último nível da heap, e inscreve-se este mesmo elemento na raiz da heap, em substituição da raiz extraída.
2. Faz-se uma operação de **bubble-down** desta nova raiz:
 Enquanto o nó actual for de valor superior a pelo menos um dos seus filhos, troca-se sucessivamente (ao longo de um caminho descendente da heap) o valor do nó com o do menor dos seus filhos



Árvores AVL

Uma árvore AVL (Adelson-Velskii & E.M. Landis) é uma árvore binária de procura em que todos os nós satisfazem adicionalmente o seguinte invariante estrutural:

As alturas da sub-árvore da esquerda e da sub-árvore da direita diferem no máximo numa unidade: $|h_e - h_d| \leq 1$.



```

typedef int TreeEntry;
typedef enum balancefactor { LH , EH , RH } BalanceFactor;

struct treenode {
    BalanceFactor bf;
    TreeEntry entry;
    struct treeNode *left;
    struct treeNode *right;
};

typedef struct treenode *Tree;

// requires:
// (t != NULL) && (t->right != NULL)
//
Tree rotateLeft(Tree t)
{
    Tree aux = t->right;
    t->right = aux->left;
    aux->left = t;
    t = aux;
    return t;
}

Tree rotateRight(Tree t)
{
    Tree aux = t->left;
    t->left = aux->right;
    aux->right = t;
    t = aux;
    return t;
}

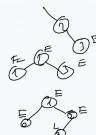


```

```

// requires:
// (t != NULL) && (t->right != NULL)
//
Tree balanceRight(Tree t)
{
    if (t->right->bf==RH) {
        // Rotacao simples, caso 3a
        t = rotateLeft(t);
        t->bf = EH;
        t->left->bf = EH;
    }
    else {
        // Dupla rotação, caso 3b
        t->right = rotateRight(t->right);
        t=rotateLeft(t);
        switch (t->bf) {
            case EH:
                t->left->bf = EH;
                t->right->bf = EH;
                break;
            case LH:
                t->left->bf = EH;
                t->right->bf = RH;
                break;
            case RH:
                t->left->bf = LH;
                t->right->bf = EH;
        }
        t->bf = EH;
    }
    return t;
}

```



```

Tree insertTree(Tree t, TreeEntry e, int *cresceu)
{
    if (t==NULL) {
        t = (Tree)malloc(sizeof(struct treenode));
        t->entry = e;
        t->right = t->left = NULL;
        t->bf = EH;
        *cresceu = 1;
    }
    else if (e > t->entry) {
        t->right = insertTree(t->right, e, cresceu);
        if (*cresceu) {
            switch (t->bf) {
                case LH: RH:
                    t->bf = EH;
                    *cresceu = 0;
                    break;
                case EH:
                    t->bf = (RH) LH;
                    *cresceu = 1;
                    break;
                case RH: LH:
                    t = balanceRight(t);
                    *cresceu = 0;
                    break;
            }
        }
    }
    else {
        t->left = insertTree(t->left, e, cresceu);
        if (*cresceu) {
            ***
        }
    }
}
return t;
}

```

Hash tables

Uma tabela de hash de capacidade cap é uma estrutura de dados física que implementa um dicionário de pares de tipo $K \rightarrow V$, e que consiste em:

1. Uma função de *hash* de tipo $h : K \rightarrow \{0, \dots, cap - 1\}$;
2. De acordo com a estratégia de resolução de colisões adoptada:

- **Open addressing:** um array com posições $\{0 \dots cap - 1\}$ de pares (k, v) , ou
- **Closed addressing:** um array com posições $\{0 \dots cap - 1\}$ de (apontadores para) listas ligadas de pares (k, v) .

Closed Addressing

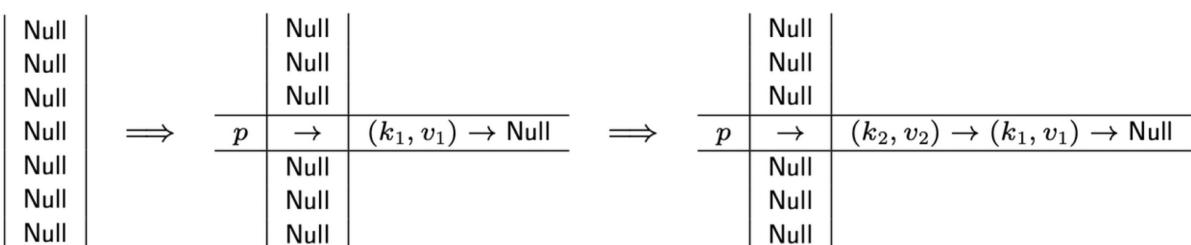
Uma solução possível para fazer “caber” vários pares chave → valor na mesma posição de um array é externalizar a informação, criando uma *lista ligada* cujo endereço inicial é guardado no array:

```
typedef struct node {  
    char key[MAXSTR];  
    ValueType info;  
    struct node * next;  
};  
typedef struct node *Hashtable[CAP];
```

Chama-se a esta implementação uma tabela encadeada (tabela com “chaining”)

A inserção na tabela faz-se agora mediante uma inserção na lista ligada apropriada.

Consideremos inserção dos pares (k_1, v_1) e (k_2, v_2) , com $h(k_1) = p$ e também $h(k_2) = p$.



Open Addressing

Ocorrendo uma colisão, procurar-se-á inserir a segunda chave numa outra posição do array, usando um método que possa ser reproduzido (nomeadamente quando se efectuar consultas)

Linear Probing

O método mais trivial de endereçamento aberto é conhecido por *linear probing*: a colisão resolve-se inserindo na posição seguinte (com circularidade) do vector.

Consideremos de novo a inserção dos pares (k_1, v_1) e (k_2, v_2) , por esta ordem, com $h(k_1) = p$ e $h(k_2) = p$.

empty, -							

⇒

empty, -							

⇒

empty, -							

Ou:

empty, -							

⇒

empty, -							

⇒

p + 1	<i>k</i>₂, <i>v</i>₂	empty, -					

Além disso, “próxima posição” deve ser de facto interpretado como “próxima posição livre”. Seja $h(k_0) = p + 1$ e $h(k_1) = h(k_2) = p$, com k_0 inserido antes de k_1 :

empty, -							

⇒

<i>p</i> + 1	<i>k</i>₀, <i>v</i>₀	empty, -					

⇒

<i>p</i>	<i>k</i>₁, <i>v</i>₁	<i>p</i> + 1	<i>k</i>₀, <i>v</i>₀	empty, -	empty, -	empty, -	empty, -

⇒

<i>p</i>	<i>k</i>₁, <i>v</i>₁	<i>p</i> + 1	<i>k</i>₀, <i>v</i>₀	<i>p</i> + 2	<i>k</i>₂, <i>v</i>₂	empty, -	empty, -

Naturalmente, a operação de consulta deve reproduzir a mesma sequência de probes utilizada na inserção.

Remoção de Chaves

Efectuemos agora as operações `insert(k1, v1)`, `insert(k2, v2)`, e `remove(k1)` por esta ordem, ainda com $h(k_1) = p$ e $h(k_2) = p$:

	empty, -
	empty, -
	empty, -
p	k_1, v_1
$p + 1$	k_2, v_2
	empty, -
	empty, -

⇒

	empty, -
	empty, -
	empty, -
p	empty, -
$p + 1$	k_2, v_2
	empty, -
	empty, -

O que sucede quando se efectuar agora uma consulta com a chave k_2 ?

Não é adequado marcar as posições onde ocorreram remoções como `empty`. Utiliza-se uma chave alternativa `removed`, que indica que uma pesquisa deve continuar para além daquela posição.

No entanto, a utilização desta chave `removed` contribui para a degradação da performance da operação de consulta. Ao fim de algum tempo já não haverá chaves `empty`, o que significa que as pesquisas de chaves inexistentes na tabela executarão todas em tempo linear.

É pois necessário proceder periodicamente a um “refrescamento” da tabela, reinicializando-a e voltando a inserir todos os pares, por forma a eliminar as chaves `removed`. Se a tabela for redimensionada frequentemente isto não será necessário, uma vez que ao redimensionar eliminam-se naturalmente as chaves `removed`.

Clustering

Um problema da estratégia de *linear probing* é a formação de clusters.

A probabilidade de cada posição ser preenchida é inicialmente dada por $r = \frac{1}{cap}$.

Revisitemos a sequência de inserções anterior, calculando a probabilidade de inserção em cada posição:

(r)	empty, -

⇒

(r)	empty, -
(r)	empty, -
(r)	empty, -
(0) p	k_1, v_1
(2*r)	empty, -
(r)	empty, -
(r)	empty, -

⇒

(r)	empty, -
(r)	empty, -
(r)	empty, -
(0) p	k_1, v_1
(0) $p + 1$	k_2, v_2
(3*r)	empty, -
(r)	empty, -

Este fenómeno de aumento da probabilidade de inserção em posições subsequentes às já preenchidas resulta na formação de “clusters”, que deterioraram localmente o comportamento das operações sobre a tabela.

Este fenómeno pode ser mitigado com a utilização de outras técnicas de *open addressing*, como *quadratic probing*.

Quadratic Probing

Em vez de fazer os probes

$p, p + 1, p + 2, p + 3 \dots$

faz-se:

$p, p + 1^2, p + 2^2, p + 3^2 \dots$

Esta técnica reduz substancialmente a formação de clusters, às custas de piorar o potencial para aproveitamento de *caching* por apresentar menor grau de localidade.