ED4. Tabelas de “hash”

Projecto Codeboard:
https://codeboard.io/projects/65989

Arrays como Dicionários

Um simples *array* de comprimento N implementa um dicionário, com o conjunto $\{0 \ldots N-1\}$ como universo de chaves.

- O tipo do array deve ser o tipo dos valores que se pretende associar às chaves
- Um valor especial pode ser usado para sinalizar que uma chave não ocorre no dicionário.

No entanto, se o universo de chaves for muito grande, torna-se incomportável a utilização directa. Por exemplo, se as chaves forem inteiros de 32 bits, teríamos de utilizar um array de comprimento 2^{32}, superior a 4 mil milhões!!!

A técnica de *hashing* permite separar o universo de chaves do conjunto de índices do array. Por exemplo, se se pretende dimensionar um dicionário com chaves de 32 bits, mas que se prevê nunca tenha uma ocupação superior a 10000, basta usar uma função h que distribua, ou disperse (*hash*) as chaves pelas posições do array:

$$h : \{0 \ldots 2^{32}-1\} \rightarrow \{0 \ldots 9999\}$$

Esta função é não-injectiva por natureza, o que significa que ocorrerão **colisões**: duas chaves poderão ser mapeadas para a mesma posição do array, e só uma pode ser inserida.

Uma característica desejável destas funções de *hash* é a **uniformidade**: todas as posições do array devem ter a mesma probabilidade de ser calculadas como resultado. Isto permitirá distribuir uniformemente a informação inserida no array, minimizando o número de colisões, bem como a formação de *clusters* em zonas específicas do array.

Quando `cap` é um número primo, a seguinte função básica comporta-se de forma razoavelmente uniforme:

```
int hash (int k, int cap) {
  return k%cap;
}
```

O desenho de uma tabela de hash implica a escolha de uma *estratégia para a resolução de colisões*, que permita inserir na tabela chaves que são à partidas mapeadas na mesma posição do array

Condicionamento de Chaves

Se as chaves não forem números naturais, devem ser previamente condicionadas, i.e. mapeadas em números naturais, novamente de forma determinista tão uniforme quanto possível.

Por exemplo, tratando-se de chaves de tipo *string* de caracteres:

```
int condition (char *s) {
  int r = 0;
  while (*s)
    r += *s++;
  return r;
}
```

Tabelas de hash

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

1. Uma função de *hash* de tipo $h : K \to \{0, \ldots cap - 1\}$;

2. De acordo com a estratégia de resolução de colisões adoptada:

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

O desenho de uma tabela de hash pretende sempre equilibrar a eficiência espacial e temporal.

- No limite, se $cap = \#(K)$, não há colisões e as operações de inserção, pesquisa, e remoção executam em tempo constante. Mas claro, desperdiça-se potencialmente muito espaço, porque o tamanho útil da tabela poderá ser muito inferior a $\#(K)$

- Ao diminuir substancialmente o tamanho cap do array, melhora-se a gestão do espaço, às custas de piorar a performance temporal das operações

- O princípio por que se rege o desenho das tabelas é que o tempo deverá sempre ser *tendencialmente constante*

Factor de Carga e Redimensionamento

Qualquer que seja a estratégia de resolução de colisões, terá de ser mantida num valor razoavelmente baixo a taxa de ocupação da tabela, ou *factor de carga:*

$$\alpha = \frac{\#\text{chaves inseridas}}{cap}$$

Ao desenhar uma tabela deve estipular-se um valor máximo para este factor, por exemplo $\alpha_{max} = 0.8$. Quando $\alpha = \alpha_{max}$ deverá redimensionar-se a tabela, que por esta razão deverá ser implementada por um *vector dinâmico*.

Relembre-se que as operações de redimensionamento (duplicação do tamanho da tabela) executam em *tempo amortizado constante.*

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:

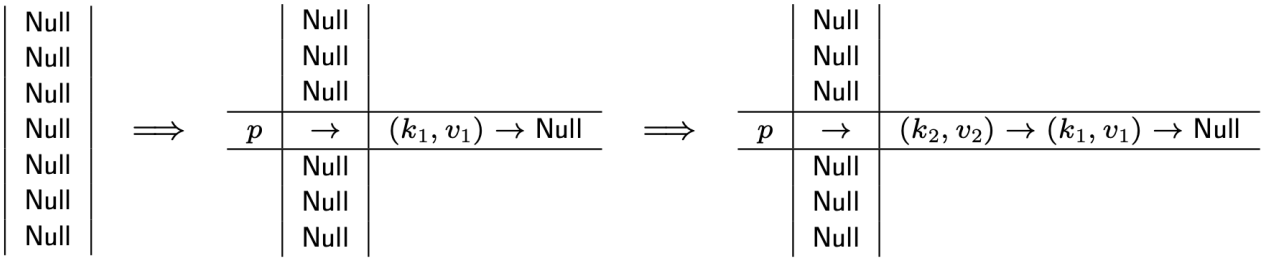
```
typedef struct node {
  char key[MAXSTR];
  ValueType info;
  struct node * next;
};
typedef struct node *Hashtable[CAP];

```

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 $h(k_2) = p$.



Pontos a reter:

- Não é suposto que estas listas cresçaam indefinidamente: apesar de em teoria

poder ser $\alpha > 1$, com um valor alto do factor de carga o tempo de execução das operações deixaria de ser “tendencialmente constante”

- As tabelas devem pois ser redimensionadas quando necessário, assegurando-se um factor de carga pequeno.

- O tempo de execução no pior caso de uma inserção ou consulta será $\Theta(1)$, desde que a função de hash seja uniforme e o factor de carga (que corresponderá ao comprimento médio das listas) pequeno.

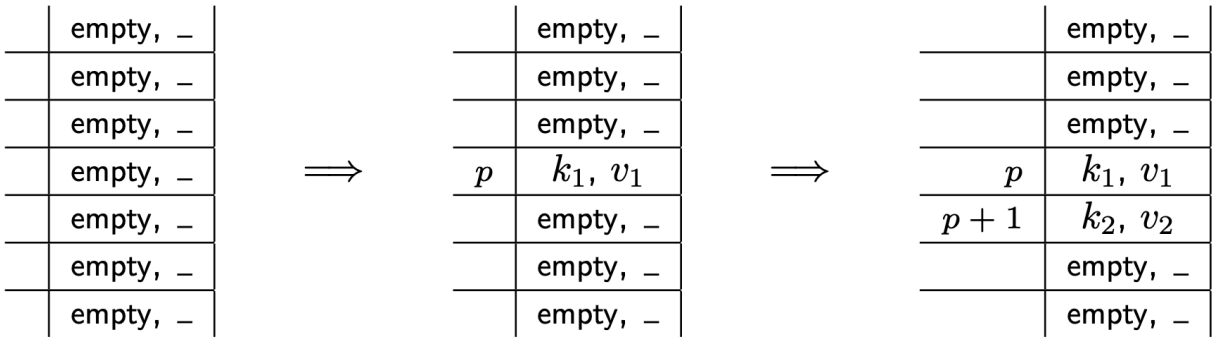
Open Addressing

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).

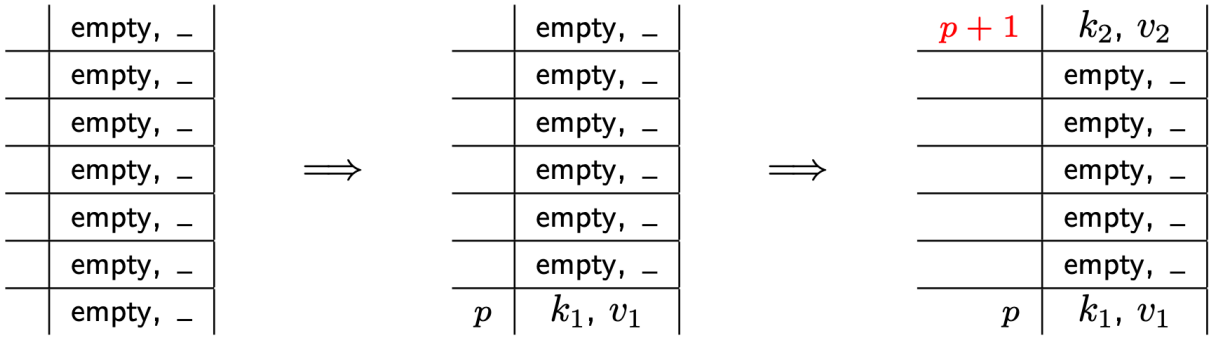
Linear Probing

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

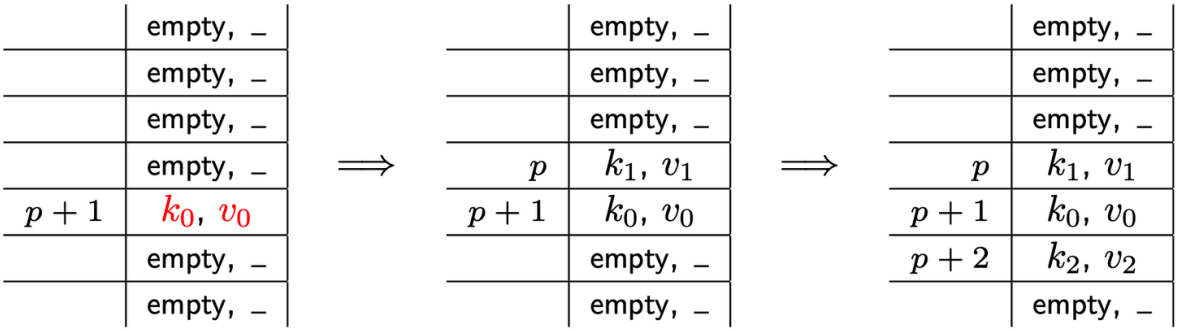
Consideremos de novo a inserção dos pares (k_1, v_1) e (k_2, v_2), por esta ordem, com $h(k_1) = p$ e $h(k_2) = p$.



Ou:



Além disso, "próxima posição" deve ser de facto interpretado como "próxima posição livre". Seja $h(k_0) = p + 1$ e $h(k1) = h(k2) = p$, com k_0 inserido antes de k_1:

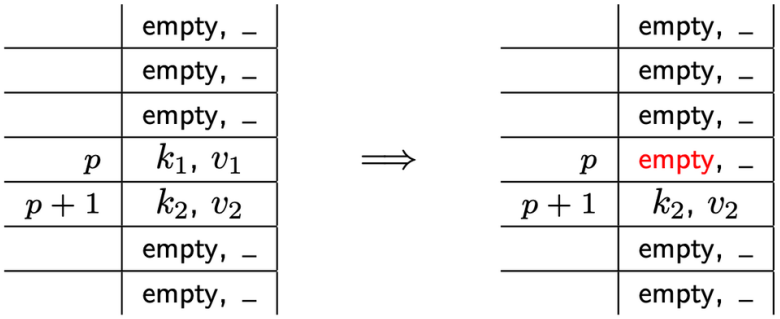


Naturalmente, a operação de consulta deve reproduzir a mesma sequência de *probes* utilizada na inserção.

> *Qual deverá ser o critério de paragem de uma operação de consulta?*

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 $h(k_2) = p$:



> *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.

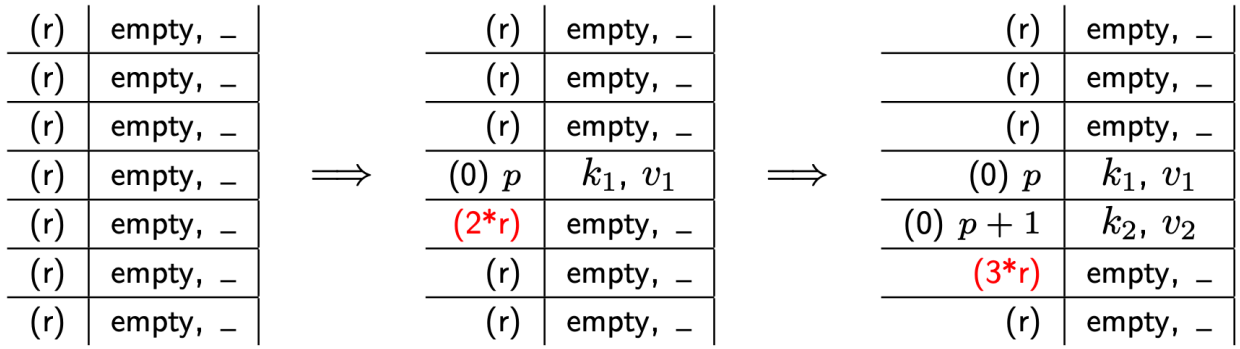
É pois necessário proceder periodicamente a um "refrescamento" da tabela, reinicializado-a e voltando a inserir todos os pares, por forma a eliminar as chaves `removed`. Se a tabela for redimensionada frequentemente isto não será necessário, uma vez que ao redimensionar eliminam-se naturalmente as chaves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



Este fenómeno de aumento da probabilidade de inserção em posições subsequentes às já preenchidas resulta na formação de "clusters", que deterioram 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

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

faz-se:

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

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

Performance

Assumindo que:

- a função de *hash* dispersa as chaves uniformemente pela tabela,
- não se formam *clusters*

é possível calcular o valor esperado (i.e. o caso médio) do número de probes efectuados nas pesquisas em caso de insucesso (chave inexistente), assim como nas inserções.

Se a tabela tiver comprimento N e factor de carga α, haverá $l = (1 - \alpha) \, . \, N$ células livres.
Como estarão espaçadas uniformemente na tabela, o número médio de células ocupadas consecutivas, que corresponde ao número médio de *probes* a efectuar será igual a N/l, ou:

$$\frac{N}{(1-\alpha).N} = \frac{1}{1-\alpha}$$

Por exemplo para $\alpha = 0.8$, teremos em média 5 *probes* em caso de insucesso.

Sendo este número independente de N, as operações são de facto realizadas em tempo constante: mantendo-se o factor de carga e aumentando o tamanho da tabela, o número de probes não aumenta, podendo no entanto aproximar-se da ordem de grandeza do tamanho da tabela. Por exemplo com $\alpha = .99$ temos uma média de 100 probes, o que poderá não ser admissível se o tamanho da tabela for apenas de algumas centenas.

Note-se no entanto que na presença de *clusters* criados por linear probing, o número de probes deixa de ser constante uma vez que o tamanho (absoluto) dos *clusters* aumenta com o tamanho da tabela.

Dados Empíricos e Comparação

O quadro seguinte contém o número de comparações efectuadas em média numa consulta de uma tabela com 900 chaves.

Factor de carga α	0.1	0.5	0.8	0.9	0.99	2.0
Consulta com sucesso, *chaining*	1.04	1.2	1.4	1.4	1.5	2.0
Consulta com sucesso, *quadratic probing*	1.04	1.5	2.1	2.7	5.2	–
Consulta com sucesso, *linear probing*	1.05	1.6	3.4	6.2	21.3	–
Consulta sem sucesso, *chaining*	0.10	0.5	0.8	0.9	0.99	2.0
Consulta sem sucesso, *quadratic probing*	1.13	2.2	5.2	11.9	126	–
Consulta sem sucesso, *linear probing*	1.13	2.7	15.4	59.8	430	–

Fonte:

Kruse R.L., Leung B.P., Tondo C.L., *Data Structures and Program Design in C*. Prentice-Hall, 2nd. ed., 1991

Observa-se que, em caso de insucesso,

- Com *chaining* o número de *probes* coincide com o factor de carga (que corresponde ao comprimento médio das listas ligadas).
- Com *open addressing* e *quadratic probing* (ou seja na ausência de *clustering*) o número de *probes* em caso de insucesso é muito próximo de $\frac{1}{1-\alpha}$.
- Com *open addressing* e *linear probing* a performance degrada-se muito.

Conclusões

- Para factores de carga razoáveis (≤ 0.8), ambas as soluções alcançam número constante de comparações nas consultas. Mas esta contagem não é o único aspecto relevante!
- *Open addressing* devidamente optimizado (*quadratic probing*, redimensionamento dinâmico para evitar factores de carga elevados) pode ser uma boa escolha, porque
  - não é penalizador em termos de espaço, ao contrário de implementações baseadas em *closed addressing / chaining*
  - apresenta vantagens em termos de *caching* (mesmo com *quadratic probing*). A grande desvantagem é a dificuldade em lidar com remoções.

- *Closed addressing / chaining é* eficiente e de programação simples, mas com um custo adicional de espaço relevante, e com as desvantagens inerentes à utilização de estruturas ligadas em termos de localidade.