Métodos de Programação II
Acetatos das aulas Teórico-Práticas

Manuel Bernardo Barbosa

May 13, 2003

Manuel Bernardo Barbosa - mbb@di.uminho.pt

O que é preciso saber para começar?

- A sintaxe básica do C.
- O que são variáveis estáticas e dinâmicas.
- O que é um apontador.
- Como é que se trabalha com apontadores em C (`*`, `->`, `NULL`, `malloc`, `sizeof`, `free`, etc).
- Vantagens de utilizar alocação dinâmica de memória: nomeadamente em ambientes de multi-tasking é vital que a memória alocada a cada processo seja o mínimo necessário, para aumentar a eficiência global do sistema.

Introdução$_{cont}$

- Diferenciar entre um tipo de dados abstracto (lista, fila, pilha) e os detalhes da sua implementação.

- A definição de um tipo de dados abstracto consiste apenas numa especificação dos elementos desse tipo, de como eles se relacionam e das operações que se podem efectuar sobre esse tipo abstracto.

- No desenvolvimento de uma aplicação deve primeiro considerar-se que tipo abstracto de dados é mais apropriado, e só depois pensar nos detalhes da sua implementação (aproximação top-down).

- Em resumo, toda a matéria coberta pela **ficha de revisões** disponível na página da disciplina: stacks, queues, listas e suas implementações em C.

Árvore Binária

- Definição: Uma árvore binária é uma estrutura vazia, ou é constituída por um nó (raiz) e por duas árvores binárias chamadas sub-árvores esquerda e direita, respectivamente.

- Note-se que, apesar da implementação ligada ser a que decorre de forma mais natural da definição, outras implementações são possíveis.

- Operações básicas sobre a estrutura de dados árvore binária:
  - Inserir
  - Remover
  - Percorrer (Traverse)

Árvore Binária$_{cont}$

- Operações de gestão:
  - Criar
  - Esvaziar
  - Obter o tamanho
  - Verificar se está vazia
- Outras operações:
  - Balancear
- Porque é que a definição de árvore binária inclui o caso vazio? O caso vazio é a condição de paragem para os algoritmos recursivos em que se baseiam as operações realizadas sobre árvores binárias.

Árvore Binária$_{cont}$

- A distinção entre sub-árvore esquerda e direita é muito importante. De facto, se analisarmos quantas árvores binárias é possível construir com dois nós, há que distinguir entre dois casos possíveis:

- Para três nós, passamos a ter cinco árvores binária possíveis, e, para quatro nós, as possibilidades são 14.

- É também importante definir a ordem pela qual se percorrem os nós de uma árvore binária e.g. quando se pretende fazer um dump do conteúdo da árvore.

Árvore Binária$_{cont}$

- Se definirmos V como a operação de ler a raiz de uma árvore binária, L como a operação de percorrer a sua sub-árvore esquerda e R como a operação de percorrer a sua sub-árvore direita, temos seis alternativas:

  VLR, LVR, LRV, VRL, RVL e RLV.

- Em geral, as alternativas em que R é efectuada antes de L não são utilizadas. Assim definem-se:
  - Preorder – VLR
  - Inorder – LVR
  - Postorder – LRV

- Um exemplo da importância destas três formas de percorrer uma árvore são as árvores de comparação resultantes de uma pesquisa binária: apenas se utilizarmos o Inorder obtemos o resultado ordenado.

Árvore Binária$_{cont}$

- Um outro exemplo são as árvores de expressões:
  - O que acontece com os operadores unários?
  - O que acontece se se percorrer Inorder? Uma expressão infix.
  - O que acontece se se percorrer Postorder? Uma expressão postfix.

- Implementação ligada:

```
typedef int TreeEntry;
struct treenode {
   TreeEntry entry;
   struct treenode *left;
   struct treenode *right;
};
typedef struct treenode *Tree;
```

Árvore Binária de Pesquisa

- Para aplicações de recuperação de informação, as implementações ligadas de listas têm o problema de não permitirem o acesso aleatório à informação armazenada: é sempre necessária uma pesquisa sequencial.

- Por outro lado, a implementação contígua de uma lista é muito ineficiente quando se pretendem fazer alterações frequentes.

- As árvores binárias são uma solução muito boa para aplicações em que é necessário combinar estes dois tipos de operações.

- De facto, utilizando árvores binárias conseguimos tempos de pesquisa na ordem O(log n), como acontece numa pesquisa binária sobre uma implementação contígua de uma lista ordenada.

Árvore Binária de Pesquisa$_{cont}$

- Simultâneamente, conseguimos obter algoritmos para inserção e remoção de entradas com tempos de execução da mesma ordem O(log n).

- As árvores binárias utilizadas para este efeito chamam-se árvores binárias de pesquisa e definem-se como árvores binárias nas quais todos os nós satisfazem as seguintes condições:
  - A entrada da raiz da sub-árvore esquerda, se existir, é menor que a entrada da raiz da árvore.
  - A entrada da raiz da sub-árvore direita, se existir, é maior que a entrada da raiz da árvore.
  - As sub-árvores são, por sua vez, árvores binárias de pesquisa.
  - Não pode haver entradas duplicadas.

Árvore Binária de Pesquisa$_{cont}$

- Em resumo: pode olhar-se para uma árvore binária de pesquisa como um tipo abstracto próprio, como um caso particular das árvores binárias, ou como uma implementação de uma lista ordenada.

- Devido às particularidades desta estrutura de dados, é costume adoptar-se a primeira visão.

- Assim, além das operações que definimos para as árvores binárias, temos de definir uma operação Treesearch (pesquisa) para este novo tipo de dados abstracto.

- Olhando para as árvores de pesquisa binária como casos particulares das árvores binárias, vemos que as operações Criar, Esvaziar, Obter o tamanho, Testar se está vazia e Traverse (nas suas três versões) são idênticas às definidas para as árvores binárias.

Árvore Binária de Pesquisa$_{cont}$

- As operações de inserção e remoção têm de ser implementadas de forma a respeitar as restrições impostas pela definição do tipo de dados.

- Finalmente, é importante manter em mente as árvores de pesquisa binária podem ser utilizadas nas mesmas aplicações para as quais se utilizam lista ordenadas.

- A ferramenta que nos permite fazer esta equiparação é herdada das árvores binárias: a operação traverse na sua versão inorder.

Árvore Binária de Pesquisa: Treesearch

- Esta operação está directamente relacionada com o algoritmo de pesquisa binária sobre uma lista ordenada.

- Começando pela raiz da árvore, procura-se o elemento desejado na sub-árvore esquerda ou direita, conforme ele seja menor ou maior que o elemento armazenado no nó actual.

- Termina quando o elemento é encontrado, ou quando se encontra uma sub-árvore vazia.

- Pode ser implementada de forma recursiva, ou de forma não recursiva.

Árvore Binária de Pesquisa: Treesearch$_{cont}$

- Em ambos os casos podemos ter tempos de pesquisa óptimos da ordem O(log n).

- A performance real do algoritmo, depende fortemente da forma da árvore, nomeadamente, da sua profundidade (não do número total de elementos).

- Assim, a forma como uma árvore deste tipo é construída e mantida determina a performance do algoritmo Treesearch.

Árvore Binária de Pesquisa: Inserção

- Esta operação vai determinar como é que elementos introduzidos numa árvore binária de pesquisa, por ordem aleatória, vão ser armazenados i.e. a forma da árvore.

- O algoritmo terá de respeitar as restrições estruturais deste tipo de árvore: um elemento inferior/superior ao elemento do nó actual terá de ser armazenado na sub-árvore esquerda/direita.

- A performance do algoritmo de inserção é equivalente à da pesquisa do elemento a inserir.

- Se a árvore não degenerar, será da ordem de O(log n).

- Pode também ser implementada de forma recursiva ou não recursiva.

Árvore Binária de Pesquisa: Inserção$_{cont}$

- A ordem pela qual os elementos são inseridos determina a forma da árvore.

- Em relação à operação de pesquisa, o pior caso ocorrerá se os elementos forem inseridos na árvore pela sua ordem natural.

- Para evitar este caso pode usar-se uma versão alternativa da operação de inserir, específica para este tipo de situação. Este algoritmo baseia-se no facto de o elemento i a ser inserido estar sempre no nível k, em que k é o expoente da maior potência de 2 que divida i.

- Uma ordenação chamada Treesort pode fazer-se inserindo os elementos de uma lista numa árvore binária de pesquisa, e depois percorrendo essa árvore usando o método inorder.

Árvore Binária de Pesquisa: Remoção

- Esta operação é mais complicada devido às restrições definidas para as relações entre os elementos armazenados.

- É necessário considerar três casos: remoção de uma folha, remoção de um nó com apenas uma sub-árvore e remoção de um nó com duas sub-árvores.

- Para a resolução do ultimo caso há várias hipóteses, a mais simples das quais pendurar uma das sub-árvores debaixo da outra. Os problemas resultantes para a estrutura da árvore são evidentes.

- Uma solução melhor será substituir o nó a remover pelo seu antecessor (ou sucessor) directo.

Árvore Balanceada (AVL)

- Calcula-se que o custo médio de performance de não utilizar uma árvore totalmente balanceada para efectuar uma pesquisa é da ordem dos 39%.

- Uma solução consiste em reestruturar a árvore uma vez que ela esteja completamente construída. Há algoritmos razoavelmente simples que permitem construir uma árvore balanceada a partir de uma lista ordenada.

- No entanto, na maior parte das aplicações, o facto de haver alterações frequentes torna este solução pouco prática.

- As árvores AVL (o nome resulta das iniciais dos seus inventores) são casos particulares de árvores de pesquisa binária em que as operações de inserção e remoção são desenhadas para manter a árvore muito próxima de um estado balanceado em cada instante.

Árvore Balanceada (AVL)$_{cont}$

- Uma árvore AVL nunca excede $(1.44 \log n)$ em altura, o que implica que, mesmo no pior caso, o tempo de pesquisa numa árvore AVL é da ordem de O(log n).

- Isto é equivalente ao caso médio numa árvore binária de pesquisa construída de forma realmente aleatória.

- Numa árvore perfeitamente balanceada, as sub-árvores de cada nó têm a mesma altura.

- Na prática, isto pode não ser possível mas, construindo a árvore cuidadosamente, é possível garantir que as sub-árvores de cada nó não diferem de mais de uma unidade nas suas alturas.

Árvore Balanceada (AVL): Definição

- Uma árvore AVL é uma árvore binária de pesquisa em que as sub-árvores esquerda e direita da raiz não diferem de mais do que uma unidade nas suas alturas.

- Por sua vez, cada uma destas sub-árvores é também uma árvore AVL.

- Cada nó numa árvore AVL está associado a um factor de balanceamento que pode tomar três valores: esquerda mais alta ($/$), alturas iguais ($-$) ou direita mais alta ($\backslash$).

- Note-se que não é necessário que todas as folhas estejam ao mesmo nível, ou mesmo em níveis adjacentes.

Árvore Balanceada (AVL): Implementação

- Implementação ligada:

```
typedef char TreeEntry;
typedef enum balancefactor { LH , EH , RH } BalanceFactor;
typedef struct treenode Treenode;
struct treenode {
   BalanceFactor bf;
   TreeEntry entry;
   TreeNode *left;
   TreeNode *right;
};
```

Árvore Balanceada (AVL): Inserção

- A operação de inserção é, na maior parte dos casos, exactamente igual à das árvores binárias de pesquisa.

- De facto, desde que a inserção não perturbe a altura da sub-árvore correspondente, o balanceamento da raiz não se altera.

- Mais do que isso, mesmo que a altura da sub-árvore correspondente se altere, em muitos casos apenas será necessário actualizar o factor de balanceamento da raiz.

- Só surgem complicações quando a inserção provoca um aumento da altura de uma sub-árvore que já é mais comprida que a sua complementar.

Árvore Balanceada (AVL): Inserção$_{cont}$

- Quando isto acontece há duas situações a considerar relativamente ao estado da sub-árvore que causa o problema. Suponhamos que se trata da sub-árvore direita:

  1. Se está desequilibrada para a direita, o problema resolve-se com uma rotação para a esquerda.
  2. Se está desequilibrada para a esquerda, é necessária uma dupla rotação (primeiro à direita e depois à esquerda).

- Note-se que a sub-árvore nunca pode estar balanceada uma vez que sabemos que a sua altura acabou de aumentar.

- No caso do problema surgir na sub-árvore esquerda a solução é simétrica.

- A operação de remoção de uma árvore AVL é bastante complexa e é proposta como exercício.

Árvore Balanceada (AVL): Inserção - 1º Caso

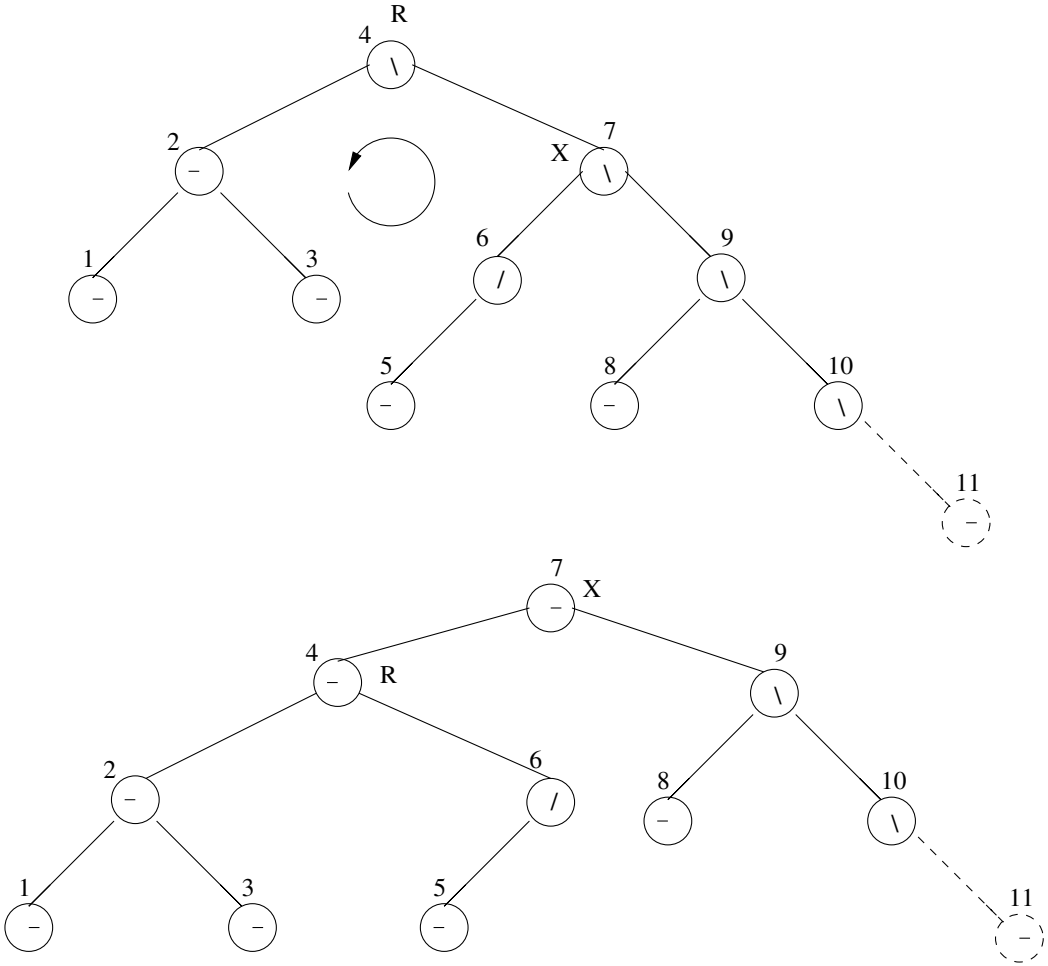
- Inseriu-se um elemento na árvore (com raiz r) e a sub-árvore da direita (com raiz x) passa a ter um desequilíbrio para a direita que causa uma violação das regras de balanceamento.

- A rotação consiste em construir uma nova sub-árvore, com raiz r, sub-árvore esquerda igual à anterior sub-árvore esquerda de r, e sub-árvore direita igual à anterior sub-árvore esquerda de x.

- A árvore AVL final tem x como raiz, a sub-árvore do ponto anterior do lado esquerdo, e , como sub-árvore direita, a original sub-árvore direita de x.

- Os nós x e r da nova árvore têm ambos um factor de balanceamento equilibrado.

Árvore Balanceada (AVL): Inserção - 1º Caso$_{cont}$



Árvore Balanceada (AVL): Inserção - 2º Caso

- Inseriu-se um elemento na árvore (com raiz r) e a sub-árvore da direita (x) passa a ter um desequilíbrio para a esquerda que causa uma violação das regras de balanceamento. A sub- árvore esquerda de x tem raiz w.

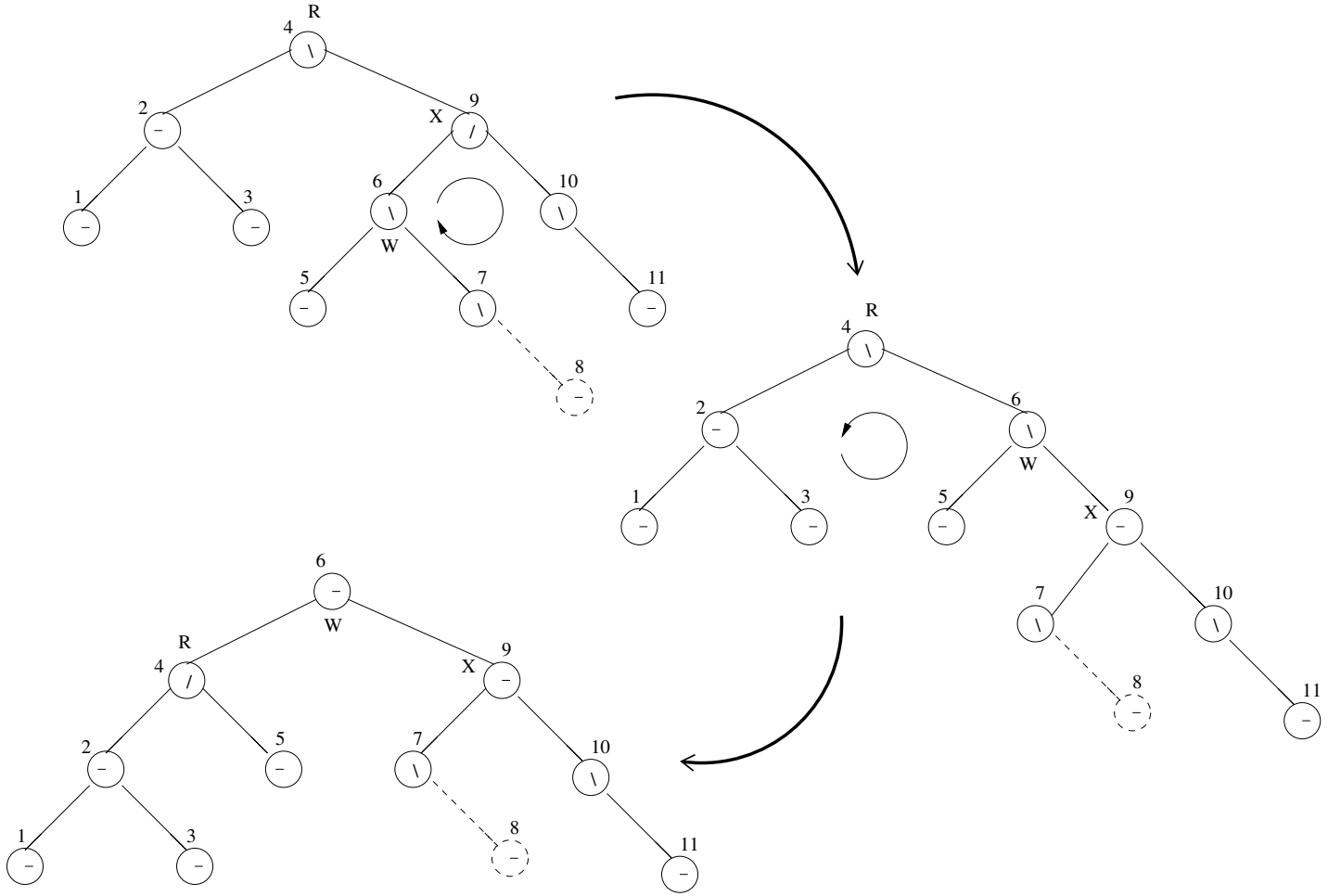
- Na primeira rotação, w passa para o lugar de x que, por sua vez, passa a constituir a raiz da sub-árvore direita de w. A sub-árvore direita original de w é pendurada do lado esquerdo de x.

- Com esta rotação, w passa a ser a raiz de uma sub-árvore desequilibrada para a direita: temos o caso anterior. Daí que seja necessário uma segunda rotação para a esquerda equivalente à descrita anteriormente.

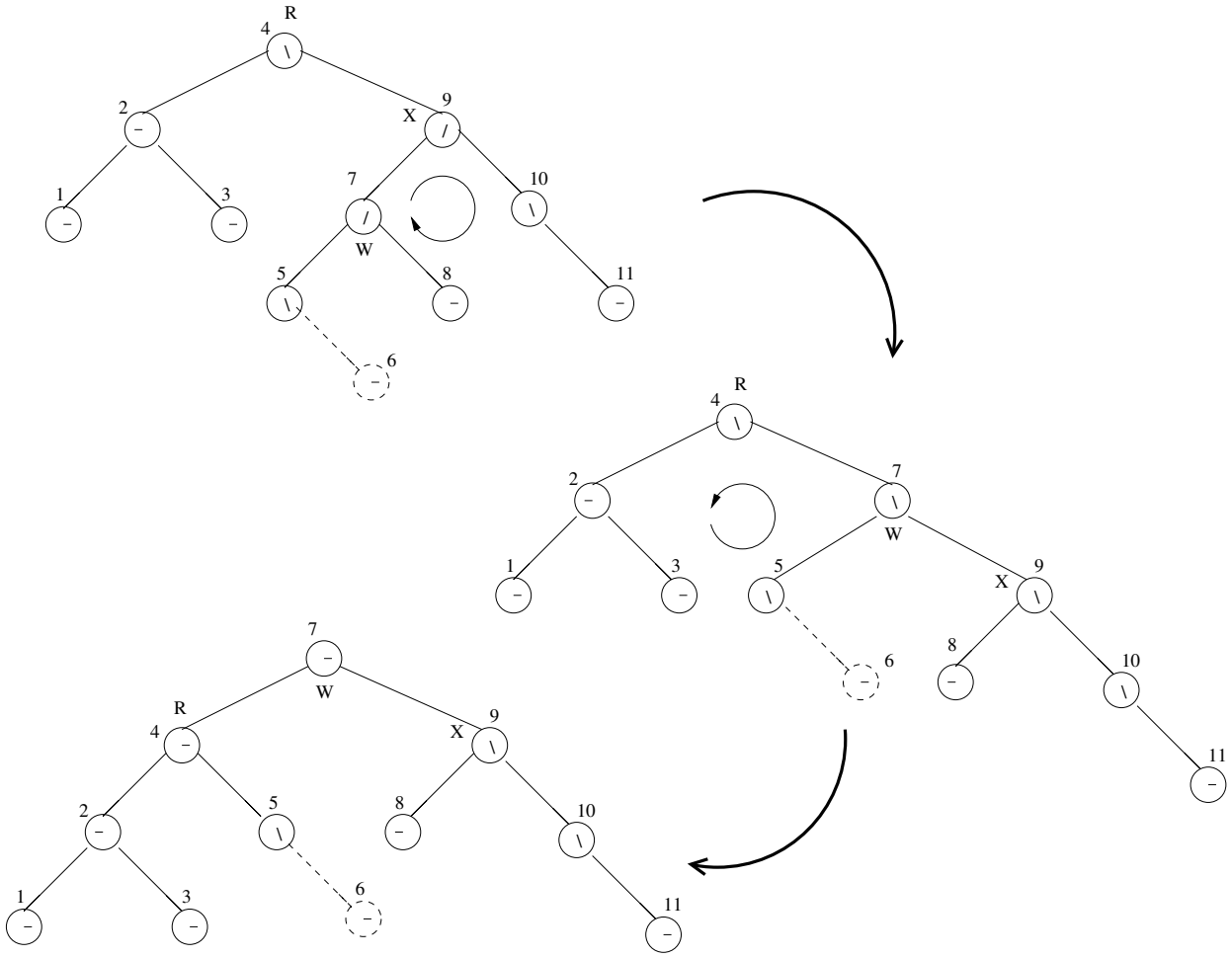
- Os novos factores de balanceamento de r e x dependem do factor de balanceamento original de w: (– apenas quando w acaba de ser criado)

w	–	\	/
r	–	/	–
x	–	–	\

Árvore Balanceada (AVL): Inserção - 2º Caso$_{cont}$



Árvore Balanceada (AVL): Inserção - 2º Caso$_{cont}$



Tabelas e Acesso a Informação

- As estruturas de dados apresentadas anteriormente têm como objectivo o armazenamento de informação, para posterior recuperação.

- No entanto, como as pesquisas se baseiam apenas em comparações sobre os valores das entradas, há um limite para a eficiência das operações de acesso à informação: não é possível fazer pesquisas com complexidade temporal inferior a O(log n).

- As estruturas de dados apresentadas nesta secção permitem quebrar este limite.

- Quando temos uma lista de N registos numerados de 1 a N, e pretendemos aceder ao registo no índice i, existe uma solução mais eficiente: armazenar os registos num array, e efectuar um acesso aleatório com base no índice do registo. Este tipo de acesso é a base do conceito de **tabela**.

Tabelas e Acesso a Informação$_{cont}$

- Definição: Uma tabela com conjunto de índices I e conjunto de entradas T é uma função $I \rightarrow T$, sobre a qual se podem efectuar as seguintes operações:
  - Aceder à tabela – calcular o valor da função para um dado índice em I.
  - Alterar a tabela – alterar a própria função, modificando o valor que é devolvido para um determinado índice em I.
  - Acrescentar à tabela – adicionar um novo valor a I e a sua imagem a T.
  - Remover da tabela – retirar um valor de I, bem como a sua imagem de T.
  - Criar/Esvaziar a tabela – a função começa por ter domínio e contradomínio vazios.

Tabelas e Acesso a Informação$_{cont}$

- Em geral há que distinguir entre a definição abstracta de Tabela e a sua implementaçã típica:
  - um array,
  - uma função de indexação que mapeia o conjunto I num conjunto intermédio de índices do array,
  - e uma função de acesso ao array que retorna o valor armazenado numa determinada entrada do array.

- Esta definição de tabela permite identificar a razão pela qual se consegue quebrar a barreira do O(log n): conceptualmente, a avaliação de uma função não depende do tamanho do seu domínio.

- Da mesma forma, o acesso a uma tabela não depende do número de elementos que nela estão armazenados (note-se que na prática isto não é totalmente verdade).

Tabelas de Hash

- O caso mais simples de uma tabela já foi apresentado na secção anterior e consiste no caso em que o conjunto I da tabela coincide com o conjunto de índices do array sobre o qual ela é implementada.

- Quando isto não é possível, é necessário definir uma função de indexação que mapeie o conjunto I no conjunto de índices do array.

- A forma mais directa de fazer isto é desenvolver uma função que faça uma correspondência de um para um entre valores destes conjuntos e.g. uma função que transforme um string num inteiro de forma injectiva.

- O problema aparece quando o conjunto I tem um número de elementos muito elevado. Além disso sabe-se muitas vezes que apenas uma fracção muito pequena dos elementos de I ocorrerão de facto (e.g. strings).

Tabelas de Hash$_{cont}$

- O caso anterior corresponde a um tipo de tabela chamada dispersa ou $sparse$.

- Uma tabela de hash consiste em utilisar uma função de indexação que mapeia vários elementos de I no mesmo índice do array. Assim, podemos alocar um array com um tamanho aceitável.

- Poderá acontecer que dois elementos de I venham a ser armazenados na mesma posição: este é um problema que tem de ser resolvido.

- No entanto, se a proporção de elementos de I que efectivamente ocorre for suficientemente pequena, e se a relação entre a dimensão deste conjunto e o tamanho do array estiver dentro de determinados parâmetros, este tipo de situação ocorrerá poucas vezes, e terá um efeito desprezável na performance do sistema.

Funções de Hash

- A função de indexação, neste caso, chama-se **função de hash**. O primeiro problema a resolver quando se quer utilizar uma tabela de hash é a escolha de uma função de hash.

- Quando ocorrem dois valores de I que são mapeados na mesma posição do array, diz-se que ocorreu uma colisão. Há diversos métodos para tratar colisões. Na implementação de uma tabela de hash, há também que escolher um destes métodos.

- Na escolha de uma função de hash devem ser asseguradas duas propriedades:
  - a função deve ser calculável de forma eficiente, e
  - deve dar origem a uma distribuição uniforme dos elementos de I pelo espaço de indexação.

Funções de Hash$_{cont}$

- Caso saibamos quais as chaves que vão ocorrer, podemos projectar uma função de hash "óptima", mas isto raramente acontece.

- Em geral, o que se faz é cortar a chave em "pedaços", "mistura-los" de várias maneiras e, desta forma, obter um índice.

- A função de hash tem de eliminar padrões que ocorram nas chaves. Há diversas técnicas que podem ser utilizadas:

  Truncation – ignorar parte da chave e utilizar a parte restante para formar o índice.

  Folding – partir a chave em diversas partes e combina-las através de adição, multiplicação ou outra operação algébrica (melhor).

  Aritmética modular – utilizar as técnicas anteriores para gerar um inteiro e calcular o resto da divisão inteira desse valor pelo tamanho do array alocado (muitas vezes um número primo).

Resolução de colisões: Open Addressing

- A forma mais simples de resolver uma coisão é, uma vez que a posição do array onde se pretende armazenar a chave está ocupada, escolher outra posição no array, de acordo com um determinado critério.

- O critério mais imediato para escolher essa nova posição é efectuar uma pesquisa sequencial, ou **linear probing**, a partir do local da colisão.

- No entanto, este método tem o problema de ser instável: quando as colisões são, mesmo que pontualmente, mais frequentes, deixa rápidamente de haver uma distribuição uniforme das chaves pelo espaço de indexação (clustering).

- Isto causa uma deterioração da performance no acesso à tabela: no limite passamos a ter uma pesquisa sequencial.

Resolução de colisões: Open Addressing$_{cont}$

- Porque não utilizar uma segunda função de hash? Isto trará poucos benefícios caso a função principal já forneça uma distribuição suficientemente uniforme.

- É necessária uma forma mais sofisticada de se escolher uma nova posição no array quando ocorre uma colisão. As três formas mais comuns de se resolver este problema são:

  – Pesquisa quadrática – a posição escolhida depois da colisão i é $(h+i^2)\%HASHSIZE$. À função i^2 chama-se função de incremento. A qualidade deste método depende de HASHSIZE, uma vez que este valor determina quais as posições passíveis de serem sondadas. Se HASHSIZE é um número primo, demonstra-se que metade das posições serão sondadas antes de se voltar à posição inicial, e que este é o melhor caso.

Resolução de colisões: Open Addressing$_{cont}$

– Incrementos dependentes da chave – em vez de fazer o incremento depender do número de colisões, faz-se depender da chave que causou a colisão. Por exemplo, quando se utiliza $\%HASHSIZE$ para calcular a função de hash, o quociente desta divisão é uma boa escolha para o incremento (e dá também uma implementação eficiente).

– Pesquisa aleatória – a posição escolhida depois da colisão i é gerada através de um gerador de números pseudo-aleatórios. Este gerador utiliza a chave como semente e deverá gerar a mesma sequência de valores para uma mesma semente. Este método é muito bom, mas pouco eficiente.

Resolução de colisões: Open Addressing$_{cont}$

- Implementação:

```
#define EMPTY NULL
typedef char *Key;
struct item {
   Key key;
   /* Aqui entrariam outros campos da tabela */
};
typedef struct item Entry;
typedef Entry HashTable[HASHSIZE];
```

- Nas remoções não podemos permitir que a cadeia de pesquisa seja quebrada: uma solução consiste em definir um valor especial (diferente de EMPTY) que assinale uma entrada removida.

Resolução de colisões: Chaining

- A utilização de espaço contíguo (um array) como base para uma tabela de hash é a opção mais natural. No entanto, se nos restringirmos a este tipo de suporte, temos duas limitações:
  - somos obrigados a utilizar a solução anterior para resolver as colisões.
  - o número de chaves inseridas está limitado ao tamanho do array.

- Uma solução mais elegante, e também mais flexível, para tratar o problema das colisões consiste em utilizar as posições do array para armazenar apontadores para as chaves armazenadas.

- Em que é que esta solução facilita a resolução de colisões? Podemos "pendurar" numa dada entrada do array mais do que uma chave: podemos guardar no array o apontador para uma lista (ligada) dos elementos cujo hash aponta para uma determinada entrada.

Resolução de colisões: Chaining$_{cont}$

- Vantagens:
  - se as entradas da tabela forem registos muito grandes, a memória reservada para o array vem muito reduzida.
  - caso ocorra um número elevado de colisões, isso não afectará o resto da tabela: apenas tornará mais lento o acesso à lista associada à entrada em conflito.
  - o número de chaves armazenadas passa a poder ser superior à dimensão do array.

- Desvantagens:
  - memória adicional para os apontadores nas listas ligadas.
  - acréscimo de complexidade inerente á gestão das próprias listas.

Resolução de colisões: Chaining$_{cont}$

- Implementação:

```
#define HASHSIZE 997
typedef char *Key;
struct item {
   Key key;
   /* Aqui entrariam outros campos da tabela */
   struct item *next;
};
typedef struct item Entry;
typedef Entry *HashTable[HASHSIZE];
```

- Como nota final sobre chaining, note-se que as remoções se tornam muito mais simples: é possível efectua-las directamente sobre as listas ligadas.

Resolução de colisões: Comparação dos Métodos

- Define-se o factor de carga de uma tabela de hash como $\lambda = n/t$, sendo n o número de elementos na tabela, e t o tamanho do array.
  - Utilizando open addressing, o factor de carga nunca pode ser superior a 1.
  - Utilizando chaining, o factor de carga não tem (teóricamente) limite superior.

- A performance dos acessos a uma tabela de hash depende **exclusivamente do seu factor de carga**:
  - **não depende do número de elementos armazenados**.

Resolução de colisões: Comparação dos Métodos$_{cont}$

- Demonstra-se que a recuperação de uma chave de uma tabela de hash que use chaining implica, aproximadamente:
  - $1 + 0.5\lambda$ comparações no caso de sucesso e,
  - λ comparações no caso de insucesso.

- No caso de se utilizar open addressing, demonstra-se que:
  - Utilizando pesquisa aleatória, o número aproximado de comparações será $(1-\lambda)^{-1}$ no caso de uma pesquisa sem sucesso, e $\lambda^{-1}ln((1-\lambda)^{-1})$ no caso de uma pesquisa com sucesso.
  - Utilizando pesquisa sequencial, o número aproximado de comparações será $0.5(1 + (1 - \lambda)^{-2})$ no caso de uma pesquisa sem sucesso, e $0.5(1 + (1 - \lambda)^{-1})$ no caso de uma pesquisa com sucesso.

Resolução de colisões: Comparação dos Métodos$_{cont}$

- Número esperado de comparações para pesquisas bem sucedidas

Factor de Carga	0.10	0.50	0.80	0.90	0.99	2.00
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open Addressing (aleatório)	1.05	1.4	2.0	2.6	4.6	–
Open Addressing (linear)	1.06	1.5	3.0	5.5	50.5	–

- Número esperado de comparações para pesquisas sem sucesso

Factor de Carga	0.10	0.50	0.80	0.90	0.99	2.00
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open Addressing (aleatório)	1.1	2.0	5.0	10.0	100	–
Open Addressing (linear)	1.12	2.5	13	50	5000	–

Comparação dos Métodos: Conclusões

- Aceder a uma tabela de hash com 40000 entradas, implementada sobre um array com 20000 posições, é equivalente a aceder a uma tabela com 40 entradas, implementada sobre um array com 20 posições.

- O chaining é uma solução consistentemente mais eficiente no que diz respeito ao número de comparações, mas . . .
  - Como a busca numa lista ligada é mais lenta que a pesquisa num array, um menor número de comparações pode não implicar uma pesquisa mais rápida.
  - Em situações em que os registos da tabela são de tamanho relativamente grande, o tempo necessário para efectuar uma comparação torna-se mais elevado e, nestes casos, a performance do chaining é efectivamente superior.

Comparação dos Métodos: Conclusões$_{cont}$

- O chaining é também muito melhor nas pesquisas sem sucesso:
  - para entradas cujos valores de hash nunca ocorreram, apenas é necessário fazer uma comparação com um apontador nulo.
  - no open addressing o número de comparações necessárias neste caso depende do factor de carga do array.

- Entre as duas opções consideradas para o open addressing, pode concluir-se que
  - apesar de a solução aleatória ser melhor, só existem diferenças significativas no caso de pesquisas falhadas.
  - Assim, para aplicações em que se prevê um número reduzido de pesquisas sem sucesso, a simplicidade da solução sequencial torna-a, muitas vezes, preferível.

Grafos

- Um $grafo$ G consiste num conjunto V, cujos membros são chamados os $vértices$ ou $nós$ de G, e num conjunto E de pares de vértices de V.

- Estes pares são os $arcos$ ou $arestas$ de G. Se eles são ordenados, então os arcos são orientados, e o grafo diz-se $orientado$. Caso contrário, o grafo é $não\ orientado$.

- A forma natural de visualizar um grafo é representar os seus vértices por pontos ou círculos, e os seus arcos por linhas que interligam os vértices.

- Os grafos são muito importantes porque permitem modelizar muitos tipos de processos e estruturas e.g. cidades e as estradas que as interligam, componentes num circuito electrónico, etc.

Grafos$_{cont}$

- Grafos não orientados:
  - Dois vértices num grafo não orientado dizem-se $adjacentes$ se existe um arco do primeiro para o segundo.
  - Um $caminho$ é uma sequência de vértices adjacentes.
  - Um $ciclo$ é um caminho, com pelo menos três vértices, em que o último vértice do caminho é adjacente ao primeiro.
  - Um grafo diz-se $ligado$ se existe um caminho que ligue cada vértice a cada um dos restantes.
  - Estritamente falando, não são admitidos arcos múltiplos entre dois vértices, nem arcos que liguem um determinado vértice a si próprio.

Grafos$_{cont}$

- Para os grafos orientados podemos definir conceitos equivalentes:
  - Introduz-se a restrição de que todos os arcos num caminho ou ciclo têm de ter o mesmo sentido. Deste modo também os caminhos e ciclos se dizem *orientados*.
  - Um grafo orientado diz-se *fortemente ligado* caso exista um caminho orientado que ligue cada nó a cada um dos restantes.
  - Se, ao olharmos para um grafo orientado, ignorarmos os sentidos dos arcos, e disso resultar um grafo ligado, então o grafo orientado diz-se *fracamente ligado*.
  - Note-se que neste caso podem haver dois arcos a interligar os mesmos vértices, desde que tenham sentidos opostos. Na realidade um par de arcos com estas características é equivalente a um arco não orientado entre os dois vértices.

Implementações de Grafos

- A escolha de uma representação computacional para a estrutura matemática de um grafo implica dois níveis de decisão:
  - Qual a estrutura de dados abstracta que vai ser utilizada para armazenar a informação associada (conjuntos, listas ou tabelas),
  - Que implementação dessa estrutura de dados é que vai ser utilizada.

- Representação por conjuntos:
  - A representação que mais directamente deriva da definição de grafo é uma representação baseada em conjuntos.
  - Em vez de se considerarem apenas os conjuntos dos vértices e arcos, divide-se este último em sub-conjuntos: um por cada vértice.
  - Note-se que é equivalente armazenar o conjunto dos arcos ligados a um vértice ou armazenar o conjunto dos vértices que lhe estão adjacentes. De facto, é esta a solução adoptada na maior parte das vezes.

Implementações de Grafos$_{cont}$

- Implementação de conjuntos:
  - Há duas maneiras de implementar conjuntos em C:

    Bitstrings A cada possível elemento do conjunto é associado um valor boleano que indica se, de facto, esse valor ocorre no conjunto ou não. Desta implementação de conjuntos deriva uma implementação de grafos denominada *Tabelas de Adjacência*.

    Listas Os elementos do conjunto são armazenados numa lista. Desta implementação de conjuntos deriva uma implementação de grafos denominada *Listas de Adjacência*.

Implementações de Grafos: Tabelas de Adjacência

```
typedef enum {False=0, True=1 } Boolean;
typedef Boolean AdjacencyTable[MAXVERTEX][MAXVERTEX];
typedef struct graph {
  int n;
  AdjacencyTable A;
} Graph;
```

- A interpretação da tabela é simples: A[v][w] é verdadeiro se e só se existe um arco orientado de v para w.

- Note-se que um grafo não orientado é um caso particular de grafo orientado em que a matriz é simétrica.

Implementações de Grafos: Listas de Adjacência

- Implementação ligada:

```
typedef struct vertex {
  int number;
  struct edge *firstedge;
  struct vertex *nextvertex;
} Vertex;
typedef struct edge {
  struct vertex *endpoint;
  struct edge *nextedge;
} Edge;
typedef Vertex * Graph;
```

  - Todos os conjuntos são implementados como listas ligadas: mais flexível mas código menos claro, e não há acesso aleatório aos nós.

Implementações de Grafos: Listas de Adjacência$_{cont}$

- Implementação contígua:

```
typedef int AdjacencyList[MAXVERTEX];
typedef struct graph {
  int n;
  int valence[MAXVERTEX];
  AdjacencyList A[MAXVERTEX];
} Graph;
```

  - Define-se *valência* de um vértice como o número de vértices que lhe são adjacentes.
  - O array de valências determina que posições são válidas em cada array to tipo AdjacencyList.

Implementações de Grafos: Listas de Adjacência$_{cont}$

- Implementação mista:

```
typedef struct edge Edge;
struct edge {
  int endpoint;
  Edge *nextedge;
};
typedef struct graph {
  int n;
  Edge *firstedge[MAXVERTEX];
} Graph;
```

Travessia de Grafos e Aplicações

- Fazer a travessia de um grafo significa seguir sistemáticamente os arcos desse grafo, de forma a visitar os seus vértices.

- Um algoritmo de travessia pode fornecer informação importante sobre a estrutura de um grafo; por esta razão é este o primeiro passo de outros algoritmos de análise de grafos em que a informação estrutural é necessária.

- Há ainda uma série de algoritmos de análise de grafos que são versões modificadas dos algoritmos de travessia.

- De uma forma ou de outra, os algoritmos de travessia são centrais no campo dos algoritmos de análise de grafos.

Travessia Breadth First

- Dado um grafo G e um vértice s específico, chamado $fonte$, este algoritmo explora os arcos de G para descobrir cada vértice que é alcançável a partir de s.

- Os vértices são descobertos de forma uniforme em relação à largura do grafo i.e. descobre todos os vértices a uma distância k de s antes de descobrir os vértices à distância $k+1$.

- Este algoritmo calcula:
  - a distância d (o menor número de arcos que o separam de s) para cada vértice descoberto;
  - uma breadth first tree, com raiz s, que armazena o caminho mais curto para todos os vértices que podem ser alcançados a partir de s. Esta árvore é, na realidade, um sub-grafo de G.

Travessia Breadth First$_{cont}$

- A breadth first tree é armazenada como uma lista π contendo o antecessor (o pai) de cada vértice. Para s, o antecessor é vazio (NIL).

- Para armazenar os vértices que estão na fronteira é utilizada uma fila (queue) como estrutura de dados auxiliar.

- Para controlar o desenrolar da travessia, é necessário marcar os vértices como $White, Gray$ ou $Black$:

  White quando o vértice ainda não foi descoberto.
  Gray quando o vértice é descoberto pela primeira vez (representa a fronteira).
  Black quando todos os vértices que lhe são adjacentes foram já descobertos.

Travessia Breadth First$_{cont}$

```
BreadthFirstSearch(G,s)
1  for each vertex u ∈ V[G] − {s}
2        color[u] = White
3        d[u] = ∞
4        π[u] = NIL
5  color[s] = Gray
6  d[s] = 0
7  π[s] = NIL
8  Q ← ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11       u =DEQUEUE(Q)
12       for each v ∈ Adj[u]
13            if color[v] = White
14                 color[v] = Gray
15                 d[v] = d[u] + 1
16                 π[v] = u
17                 ENQUEUE(Q, v)
18       color[u] = Black
```

Travessia Breadth First$_{cont}$

- A função seguinte escreve o caminho mais curto entre s e outro vértice v utilizando os resultados do algoritmo anterior:

```
PrintPath(G,s,v)
1   if v = s
2     print s
3   else
4     if π[v] = NIL
5       print "No path from" s "to" v "exists."
6     else
7       PrintPath(G, s, π[v])
8       print v
```

Travessia Depth First

- A estratégia deste algoritmo consiste em explorar o grafo em profundidade i.e. quando o vértice actual tem um arco que não foi explorado, é esse o arco que é escolhido a seguir.

- Quando não é possivel ir mais longe, o algoritmo faz back-tracking (retrocede) até ao último vértice que ainda tem arcos por explorar, e prossegue a partir desse ponto).

- Ao contrário do algoritmo anterior, a busca é exaustiva: para assegurar que todos os vértices são percorridos, o algoritmo tenta utilizar todos os vértices como fonte da pesquisa, de forma sequencial.

- Esta modificação deve-se ao facto de a utilização típica dos resultados implicar este tipo de busca exaustiva e.g. ordenação topológica.

Travessia Depth First$_{cont}$

- Assim, em vez de termos uma árvore como resultado da travessia, passamos a ter um subgrafo de G que é uma depth first forest: uma floresta de depth first trees (uma para cada fonte utilizada).

- Tal como no caso anterior, os nós são pintados de $White$, $Gray$ e $Black$ à medida que vão sendo descobertos.

- Em vez da distancia à fonte, e como resultados adicionais, este algoritmo armazena para cada vértice
  - um $time\text{-}stamp$ d do momento em que foi descoberto pela primeira vez i.e. do momento em que foi pintado de $Gray$,
  - e outro $time\text{-}stamp$ f do momento em que todos os vértices que lhe são adjacentes foram já explorados i.e. do momento em que foi pintado de $Black$.

Travessia Depth First$_{cont}$

- A implementação mais imediata da travessia depth first baseia-se numa função recursiva:

```
DepthFirstSearch(G)
1  for each vertex u ∈ V[G]
2       color[u] = White
3       π[u] = NIL
4  time = 0
5  for each vertex u ∈ V[G]
6       if color[u] = White
7          DFSVISIT(u)
```

```
DFSVISIT(u)
1  color[u] = Gray
2  time = time + 1
3  d[u] = time
4  for each v ∈ Adj[u]
5       if color[v] = White
6          π[v] = u
7          DFSVISIT(v)
8  color[u] = Black
9  time = time + 1
10 f[u] = time
```

Travessia Depth First$_{cont}$

- A depth first forest reflecte a sequência de chamadas recursivas. As listas de $time\text{-}stamps$ permitem reconstruir essa sequência em termos temporais.

- Olhando para os $time\text{-}stamps$ d e f de dois vértices u e v, v é um descendente de u se e só se

$$d[u] < d[v] < f[v] < f[u]$$

- Note-se que os resultados podem ser diferentes consoante a ordem pela qual os vértices são examinados; no entanto isto, em geral, não causa problemas na utilização deste algoritmo.

Aplicação: ordenação topológica

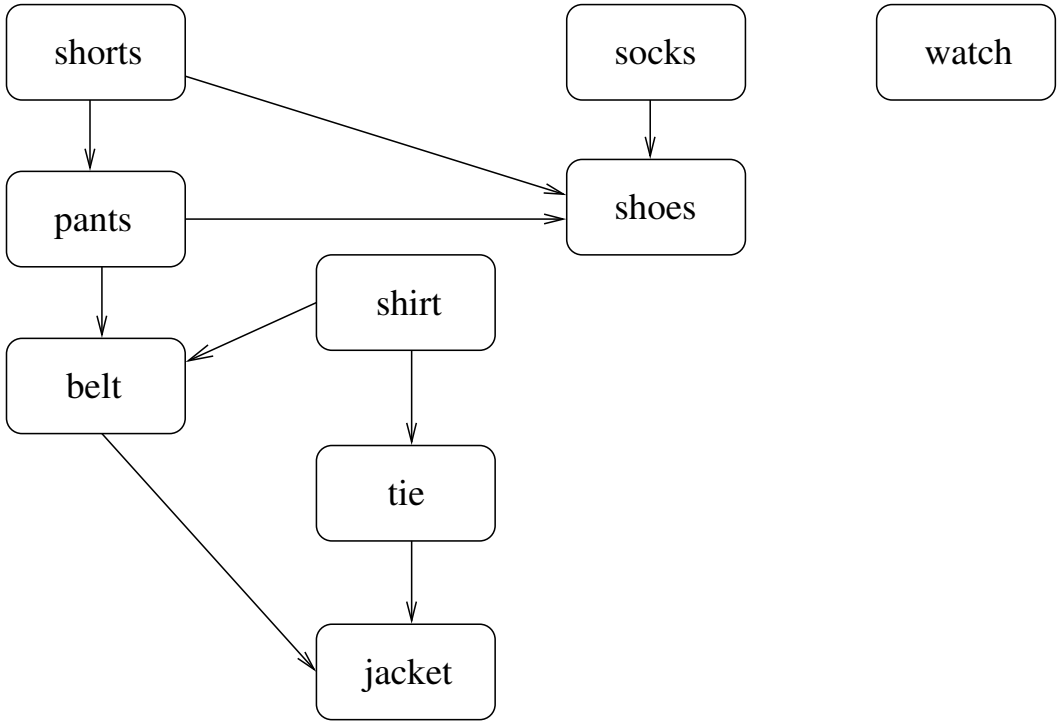
- A ordenação topológica é uma operação que se define para um caso particular dos grafos orientados, que são os grafos acíclicos ou $dags$.

- Este tipo de grafo é utilizado em muitas aplicações para representar, por exemplo, a dependência ou precedência de eventos.

- Uma ordenação topológica de um grafo G produz uma ordenação línear dos seus vértices tal que, se (u, v) é um arco de G, então u aparece antes de v na ordenação.

- Um exemplo será um grafo cujos nós represntam peças de roupa, e os arcos orientados (u, v) indicam que a peça representada pelo nó u tem de ser vestida antes da peça representada pelo nó v.

Aplicação: ordenação topológica$_{cont}$



- Uma ordenação topológica válida para este caso seria: socks, shorts, pants, shoes, watch, shirt, belt, tie, jacket.

Aplicação: ordenação topológica$_{cont}$

- O seguinte algoritmo de ordenação topológica baseia-se directamente no algoritmo de travessia depth-first apresentado anteriormente:

  TopologicalSort(G)
  1 Call DepthFirstSort(G) to compute finishing times $f[v]$ for each vertex v
  2 As each vertex is finished, insert it onto the front of a linked list
  3 **return** the linked list of vertices

Árvore Geradora Mínima (Algoritmo de Prim)

- Um grafo pesado (weighed graph) é um grafo em que cada arco tem associado um peso. Esse peso pode representar uma série de coisas: distâncias entre locais, custos, capacidades, etc.

- Uma Árvore Geradora é uma árvore (um subgrafo acíclico) que contém todos os vértices do grafo original. Essa árvore diz-se Mínima caso a soma dos pesos de todos os seus arcos ser mínimo.

- Sempre que se pretende encontrar a forma mais barata de interligar um conjunto de terminais (cidades, terminais eléctricos, etc), uma solução será a Árvore Geradora Mínima (AGM) de um grafo pesado com custos.

- Se o grafo não é ligado, então não terá uma Árvore Geradora, e muito menos uma AGM.

Árvore Geradora Mínima (Algoritmo de Prim)$_{cont}$

- O algoritmo de Prim calcula uma AGM da seguinte forma:

1. Escolhe-se um vértice qualquer que será o primeiro a pertencer à AGM.
2. Dividem-se os vértices do grafo em três partes:
   - **Vértices da árvore** são os pertencentes à AGM.
   - **Vértices da orla** são aqueles que, não pertencendo à AGM, estão adjacentes a um vértice que pertence à AGM.
   - **Vértices desconhecidos** são os restantes.
3. Escolhe-se um novo vértice e um novo arco para adicionar à AGM sendo o critério de escolha o seguinte:
   - o arco escolhido é o de menor peso com origem num dos nós da árvore e destino num dos nós da orla;
   - o vértice escolhido é o vértice de destino desse mesmo arco.
4. O procedimento prossegue do passo 2 até que não seja possível adicionar mais vértices à AGM.

Árvore Geradora Mínima (Algoritmo de Prim)$_{cont}$

- A eficiência do algoritmo é aumentada da seguinte forma:
  - Em cada iteração basta armazenar, para cada nó da orla, o arco de menor peso que o liga á AGM - **arcos candidatos**.
  - Cada vez que se adiciona um novo nó e um novo arco à AGM é necessário actualizar a lista dos arcos candidatos. Exploram-se os arcos que saem do ultimo nó a ser adicionado à AGM:
    * apenas estes poderão fornecer uma melhor solução para ligar a AGM a um nó que já pertença à orla - neste caso subsitui-se o arco candidato correspondente pelo novo arco (de menor peso).
    * apenas estes poderão levar a nós que não estavam préviamente na orla - acrescenta-se cada novo nó à orla, e armazena-se como arco candidato correspondente o arco que permitiu descobrir esse nó.

- O algoritmo seguinte recebe como parâmetro $G = (V, E)$

Árvore Geradora Mínima (Algoritmo de Prim)$_{cont}$

```
1   V' = x , T' = ∅ , stuck = 0
2   while (V' ≠ V ∧ !stuck)
3         for (y ∈ orla, y adjacente a x)
4             if (weight(x,y) < weight(arco candidato de y))
5                 substituir arco candidato de y por (x,y)
6         for (y ∉ orla ∧ y ∉ V', y adjacente a x)
7             colocar y na orla
8             marcar (x,y) arco candidato
9         if ( Não ha arcos candidatos)
10          stuck = 1
11        else
12          escolher arco candidato (u,v) de custo minimo
13          x = v , V' = V' ∪ x , T' = T' ∪ (u,v)
14          remover x da orla e desmarcar (u,v) como candidato
```

Árvore Geradora Mínima (Algoritmo de Prim)$_{cont}$

- Em cada iteração, x é o último vértice a ter sido adicionado à AGM.

- O funcionamento do algoritmo implica manter as seguintes estruturas de dados:
  - uma lista dos nós já incluidos na AGM (V');
  - para cada nó na AGM, o caminho que o algoritmo percorreu até o adicionar à árvore (equivalente a T');
  - uma lista dos nós que estão na orla (orla);
  - para cada nó da orla, o arco de menor peso que o liga a um nó da AGM i.e. o seu arco candidato;

- Uma optimização adicional consegue-se transformando os dois ciclos **for** num só. Uma implementação optimizada terá um tempo de execução, no pior caso, de $O(|E| + |V|^2)$.

Árvore Geradora Mínima (Algoritmo de Prim)$_{cont}$

```
1   V' = x , T' = ∅ , stuck = 0
2   while (V' ≠ V ∧ !stuck)
3         for (y adjacente a x ∧ y ∉ V')
4             if (y ∈ orla)
5                 if (weight(x,y) < weight(arco candidato de y))
6                     substituir arco candidato de y por (x,y)
7             else
8                     colocar y na orla e marcar (x,y) arco candidato
9         if ( Não ha arcos candidatos)
10          stuck = 1
11        else
12          escolher arco candidato (u,v) de custo minimo
13          x = v , V' = V' ∪ x , T' = T' ∪ (u,v)
14          remover x da orla e desmarcar (u,v) como candidato
```

Shortest Path (Algoritmo de Dijkstra)

- Outro problema que muitas vezes se coloca quando se lida com grafos pesados é o de encontrar o caminho mais curto entre dois vértices.

- O algorítmo anterior, apesar de calcular uma árvore geradora de peso mínimo, **não encontra o caminho mais curto entre os nós do grafo**.

- Isto deve-se ao facto de a escolha dos arcos candidatos ser feita com base num critério local: o arco escolhido é o arco de menor peso que liga a árvore a cada vértice da orla.

- Este critério minimiza o peso total da árvore geradora, mas ignora a distância a que o antecessor de cada vértice da orla está da fonte.

- Na realidade, no algoritmo de Prim não existe sequer a noção de nó fonte: o primeiro nó da AGM é escolhido à sorte.

Shortest Path (Algoritmo de Dijkstra)$_{cont}$

- O algoritmo de Dijkstra é uma versão alterada do algoritmo de Prim, que calcula o caminho mais curto entre dois nós:
  - O algoritmo passa a ser parametrizado com dois vértices do grafo: o nó de origem e o nó de destino, entre os quais pretendemos encontrar o caminho mais curto.
  - Para cada vértice da orla ou da árvore passa a ser necessário armazenar a distância a que ele se encontra da origem.
  - Os arcos candidatos passam a ser escolhidos de acordo com um critério diferente: é escolhido do arco que fornece o caminho mais curto da origem até ao nó da orla correspondente.
  - O algoritmo termina quando o nó de destino é adicionado à árvore.

- Tal como no caso anterior, o algoritmo seguinte recebe como parâmetro um grafo $G = (V, E)$.

Shortest Path (Algoritmo de Dijkstra)$_{cont}$

1 $V' = v$, $x = v$, $distance(x) = 0$, $T' = \emptyset$, $stuck = 0$
2 **while** $(x \neq w \wedge !stuck)$
3 **for** (y adjacente a x $\wedge$ $y \notin V'$)
4 **if** $(y \in orla)$
5 **if** $(distance(x) + weight(x, y) < distance(y))$
6 substituir arco candidato de y por (x, y)
7 $distance(y) = distance(x) + weight(x, y)$
8 **else**
9 colocar y na orla e marcar (x, y) arco candidato
10 $distance(y) = distance(x) + weight(x, y)$
11 **if** (não ha arcos candidatos) **then** $stuck = 1$
12 **else**
13 escolher arco candidato (u, v) com menor $distance(v)$
14 $x = v$, $V' = V' \cup x$, $T' = T' \cup (u, v)$
15 remover x da orla e desmarcar (u, v) como candidato