C103 Algoritmos e Estruturas de Dados II

Carlos Alberto Ynoguti

16 de outubro de 2018

Sumário

Capítulo 1

Ordenação de Dados

No nosso dia-a-dia é comum vermos as vantagens de manipular um conjunto de dados previamente classificado, como por exemplo, procurar o telefone de uma determinada pessoa em uma lista telefônica. Já pensou se a lista não estivesse em ordem alfabética?

Neste capítulo vamos apresentar alguns dos algoritmos mais utilizados para fazer esta tarefa, bem como uma análise comparativa entre os mesmos para que você possa escolher o melhor para a aplicação em que vai ser utilizado. Para isto, você tem que ter três coisas em mente: a quantidade de dados a ser processada, o tempo necessário para a classificação, e finalmente, o tempo necessário para desenvolver o programa.

O que? O tempo de desenvolvimento também entra na conta? Você tá maluco? Pirou de vez? Calma! Eu explico: suponha que você está desenvolvendo um programa que precise ordenar um vetor pequeno (100 elementos). Qualquer método neste caso funciona muito bem, de forma rápida e eficiente. Então não vale a pena gastar tempo desenvolvendo esta parte. Gaste sua energia no resto do programa. Entretanto, você deve tomar cuidado com isso: muitos programadores usam esta desculpa para esconder que não conhecem métodos melhores e mais eficientes. Não seja um deles. Por outro lado, se temos que ordenar vetores grandes (algo como milhões de elementos), devemos implementar algoritmos altamente eficientes para poder dar conta do recado.

1.1 Movimentação dos dados

A sequência de entrada é em geral um vetor com `n` elementos, mas existem outras possibilidades de estruturas de dados, como por exemplo as listas ligadas. Na prática, os elementos a serem ordenados são trabalhados como um conjunto de dados denominado registro, em vez de serem manipulados de forma isolada. Assim, o conjunto de registros forma um vetor. Cada registro contém uma chave, que é o valor a ser ordenado, e demais valores, que sempre acompanham a chave. Isso significa que quando realizamos um processo de ordenação, se houver a necessidade de trocar a posição de uma chave, ocorrerá também a alteração da posição de todos os elementos do registro.

Com o objetivo de minimizar a tarefa computacional de movimentação de dados durante um processo de ordenação, observa-se que, na prática, quando os registros possuem uma grande quantidade de dados (além da chave), o processo de ordenação pode ser realizado com o uso de um vetor (ou lista) de ponteiros para os registros. Uma classificação dos métodos que leva em conta estes aspectos é apresentada abaixo:

1. **Contiguidade física**: existe uma forte característica física no processo de or-

denação, porque os elementos de um registro são movimentados de sua posição física original para sua nova posição. As Tabelas abaixo mostram um exemplo:

Tabela 1.1: Antes

	Registro chave	Demais campos		
1	4	PP	PP	PP
2	7	FF	FF	FF
3	2	GG	GG	GG
4	1	DD	DD	DD
5	3	SS	SS	SS
6	5	ZZ	ZZ	ZZ

Tabela 1.2: Depois

	Registro chave	Demais campos		
1	1	DD	DD	DD
2	2	GG	GG	GG
3	3	SS	SS	SS
4	4	PP	PP	PP
5	5	ZZ	ZZ	ZZ
6	7	FF	FF	FF

2. **Vetor indireto de ordenação:** não há movimentação das entradas das posições originais. Isso é possível porque a classificação é realizada com o apoio de um vetor-índice. É justamente este vetor que contém ponteiros para os registros reais, os quais são mantidos classificados.

Tabela 1.3: Tabela desordenada

	Registro chave	Demais campos		
1	3	LL	LL	LL
2	6	AA	AA	AA
3	5	ZZ	ZZ	ZZ
4	1	KK	KK	KK
5	4	MM	MM	MM
6	2	RR	RR	RR

Tabela 1.4: Vetor de ordenação

	Índice do vetor
1	4
2	6
3	1
4	5
5	3
6	2

3. **Encadeamento:** também não há a movimentação dos registros das posições originais. A ordem desejada é conseguida pela uso de uma lista ligada. Coloca-se um campo a mais no registro para permitir que o próximo registro sempre seja identificado. Dessa forma, não é mais necessário o vetor adicional (vetor indireto de ordenação), mas temos de utilizar um ponteiro para o primeiro elemento da lista ligada formada para a classificação desejada. Observe:

5

Ponteiro para o primeiro elemento da lista.
(o registro 5 contém a chave 1)

	Campo chave	Demais campos			next
1	3	LL	LL	LL	4
2	6	AA	AA	AA	-
3	5	ZZ	ZZ	ZZ	2
4	4	KK	KK	KK	3
5	1	MM	MM	MM	6
6	2	RR	RR	RR	1

1.2 Ordenação interna versus ordenação externa

A ordenação interna é aquela na qual todo o processo ocorre na memória principal (memória RAM), e na ordenação externa a memória auxiliar (discos, fitas, etc) é utilizada.

Tá, entendi, mas e aí? Bom, pra entender a diferença, precisamos ver como as coisas funcionam em um computador: a memória RAM é um dispositivo de armazenamento extremamente rápido, enquanto que o acesso aos dispositivos de memória secundária tais como discos, é bem mais lento. Entretanto, os discos têm uma capacidade de armazenamento muito maior que a memória RAM.

Juntando alhos com bugalhos, chegamos à seguinte conclusão: se conseguirmos fazer tudo na memória RAM (ordenação interna), nosso programa funciona muito mais rápido, mas como temos muito menos espaço na memória RAM do que no disco, temos aí uma limitação quanto ao número de dados a serem processados. Por outro lado, se o volume de dados a serem processados é muito grande, não temos escolha: temos que usar a ordenação externa e ter paciência para esperar.

Neste texto vamos abordar apenas alguns dos principais métodos de ordenação interna. Estes podem ser classificados em três grupos:

- **Ordenação por inserção**
  - Inserção direta
  - Incrementos decrescentes (*Shell sort*).
- **Ordenação por troca**
  - Método da bolha (*bubble sort*);
  - Método da troca e partição (*quick sort*).
- **Ordenação por seleção**
  - Seleção direta;
  - Seleção em árvore (*heap sort*).

1.3 Ordenação por inserção

Como o próprio nome sugere, a classificação é obtida porque os elementos são inseridos na sua posição correta, levando-se em conta os elementos já classificados. Nesta classe existem dois métodos: o de inserçao direta e o método Shell. Vamos a eles.

1.3.1 Inserção direta

Este método é o mais simples, e apresenta uma baixa eficiência. Esta não faz muita diferença quando o número de dados a serem ordenados é pequeno, mas à medida que o conjunto a ser ordenado começa a crescer, o programa fica tão lento que é melhor usar outro método se você quiser manter o emprego. De forma simplificada, este algoritmo funciona da seguinte maneira:

- Um primeiro elemento está inicialmente no vetor ordenado e os demais no vetor desordenado. Por exemplo:

Vetor ordenado	Vetor desordenado
6	3 4 5 9 8

- Retira-se o primeiro elemento do vetor desordenado. Ao colocá-lo no vetor ordenado, é realizada a devida comparação para inseri-lo na posição correta.

Vetor ordenado	Vetor desordenado
3 6	4 5 9 8

- Repete-se o processo até que todos os elementos do vetor desordenado tenham passado para o vetor ordenado.

Vetor ordenado	Vetor desordenado
6	3 4 5 9 8
3 6	4 5 9 8
3 4 6	5 9 8
3 4 5 6	9 8
3 4 5 6 9	8
3 4 5 6 8 9	

Abaixo tem-se uma sugestão de código para a função `insercaoDireta`. Esta tem por entrada um parâmetro vetorial `vetor`, junto com o número de elementos de `vetor`. Ela retorna o vetor `vetor` já ordenado de forma ascendente.

```
void insercaoDireta(int vetor[], int tamanho)
{
  int i,j; // contadores
  int chave;

  for (j=1;j<tamanho;j++)
  {
    chave = vetor[j];
    i = j-1;
    while ((i >= 0) && (vetor[i] > chave))
    {
      vetor[i+1] = vetor[i];
      i = i-1;
    }
    vetor[i+1] = chave;
  }
}
```

1.3.2 Incrementos decrescentes (*shell sort*)

Este algoritmo foi proposto por Ronald L. Shell em 1959. É considerado uma extensão do algoritmo de inserção direta, pois a diferença reside apenas no número de segmentos do vetor usado para o processo de classificação.

Na inserção dirta existe apenas um único segmento do vetor onde os elementos são inseridos ordenadamente,enquanto no método *Shell* são considerados diversos segmentos. Desta forma, a ordenação é realizada em diversos passos. A cada passo está associado um incremento `I`, o qual determina os elementos que pertencem a cada um dos segmentos. Veja:

segmento 1 vetor[0], vetor[I], vetor[2I], ...
segmento 2 vetor[1], vetor[1+I], vetor[1+2I], ...
segmento 3 vetor[2], vetor[2+I], vetor[2+2I], ...
... ...

Considerando o valor inicial do incremento `I`, podemos ter vários passos. Em cada passo realiza-se uma ordenação de cada um dos segmentos usando isoladamente a inserção direta.

Repete-se o processo, ao término de cada passo, alterando o valor do incremento `I` para a metade do valor anterior, até o momento que seja executado um passo com incremento `I=1`. Observe que para que isto funcione, `I` deve ser uma potência de 2.

Uma forma fácil de implementar isso é usar uma variável auxiliar `np` que conta o número de passos a serem executados. Assim, para `np=3`, o valor do incremento em cada passo seria:

$I = 2^{np} = 2^3 = 8$
$I = 2^{np-1} = 2^2 = 4$
$I = 2^{np-2} = 2^1 = 2$
$I = 2^{np-3} = 2^0 = 1$

Observe que o número de passos é, na verdade, `np+1`.

Exemplo 1.1. Vamos acompanhar o processo sobre o vetor abaixo, utilizando `np=2`:

- Vetor original desordenado:

1	2	3	4	5	6	7	8	9	10	11	12
17	29	42	15	21	22	47	37	52	43	27	12

- Passo 1: $I = 2^{np} = 2^2 = 4$

segmento 1

1	5	9
17	21	52

segmento 2

2	6	10
29	22	43

segmento 3

3	7	11
42	47	27

segmento 4

4	8	12
15	37	12

Aplicando a inserção direta em cada segmento:

17	21	52

22	29	43

27	42	47

12	15	37

Obtém-se o vetor

1	2	3	4	5	6	7	8	9	10	11	12
17	22	27	12	21	29	42	15	52	43	47	37

- Passo 2: $I = 2^{np-1} = 2^1 = 2$

segmento 1

1	3	5	7	9	11
17	27	21	42	52	47

segmento 2

2	4	6	8	10	12
22	12	29	15	43	37

Aplicando a inserção direta em cada segmento:

17	21	27	42	47	52

12	15	22	29	37	43

Obtém-se o vetor

1	2	3	4	5	6	7	8	9	10	11	12
17	12	21	15	27	22	42	29	47	37	52	43

- Passo 3: I = $2^{np-2} = 2^0 = 1$. Neste último passo, os elementos estão próximos das suas posições finais, o que leva a um número menor de trocas. Aplicando a inserção direta ao vetor obtido anteriormente, temos:

1	2	3	4	5	6	7	8	9	10	11	12
12	15	17	21	22	27	29	37	42	43	47	52

O código da função shell sort é apresentado abaixo. Como no caso anterior, esta função tem por entrada um parâmetro vetorial `vetor`, junto com o número de elementos de `vetor`. Entretanto, esta função necessita de um argumento adicional, que é o número de passos. Ela retorna o vetor `vetor` já ordenado de forma ascendente.

```
void shellsort(int x[], int tamanho, int nPassos)
{
  int incr,i,j,k,span,y;
  int *incrementos;

  incrementos = new int[nPassos];

  for (i=0;i<=nPassos;i++)
    incrementos[i] = (int)round(exp((nPassos-i)*log(2.0)));

  for (incr=0;incr<=nPassos;incr++)
  {
    // span e o tamanho do incrementos
    span = incrementos[incr];
    for (j=span;j<tamanho;j++)
    {
      y = x[j];
      for (k=j-span;k>=0 && y<x[k];k-=span)
        x[k+span] = x[k];
      x[k+span] = y;
    }
  }
  delete [] incrementos;
}
```

1.4 Ordenação por troca

No processo de ordenação por troca, a idéia é fazer uma varredura no vetor como um todo. No momento em que encontrarmos dois elementos fora de ordem, eles trocam de lugar. Existem dois métodos principais que usam esta idéia: o método da bolha (*bubble sort*) e o método da partição (*quick sort*). A diferença entre estes métodos está na escolha dos pares de elementos que vão ser comparados. Vamos a eles.

1.4.1 Método da bolha (*bubble sort*)

É um método bastante trivial e, em geral, bastante lento. Vamos apresentá-lo, e ver se você descobre porque ele recebe este nome tão engraçado:

- Compara-se o primeiro elemento do vetor com o segundo.
- Se estiverem fora do lugar, trocam de posição.
- Depois, compara-se o segundo com o terceiro, e assim por diante até chegar ao fim do vetor.
- Nesta primeira varredura, verifica-se que o maior elemento já está em sua posição correta.
- Volta-se ao início do vetor e repete-se o processo. Ao fim desta segunda varredura, os dois maiores elementos do vetor estão em suas posições corretas.
- Novas varreduras são feitas até que todo o vetor esteja ordenado.

```
void bubbleSort(int vetor[], int tamanho)
{
  int i,j; // contadores
  int trab;
  bool troca;
  int limite;

  troca = true;
  limite = tamanho-1;
  while (troca)
  {
    troca = false;
    for (i=0;i<limite;i++)
      if (vetor[i] > vetor[i+1])
      {
        trab = vetor[i];
        vetor[i] = vetor[i+1];
        vetor[i+1] = trab;
        j = i;
        troca = true;
      }
    limite = j;
  }
}
```

Exemplo 1.2. Vamos ver como isso funciona para o vetor abaixo:

Original	510	95	525	70	920	180	900	
Passo 1	**510**	**95**	525	70	920	180	900	
	95	**510**	**525**	70	920	180	900	
	95	510	**525**	**70**	920	180	900	
	95	510	70	**525**	**920**	180	900	
	95	510	70	525	**920**	**180**	900	
	95	510	70	525	180	**920**	**900**	
	95	510	70	525	180	900	**920**	920 no lugar
Passo 2	**95**	**510**	70	525	180	900	920	
	95	**510**	**70**	525	180	900	920	
	95	70	**510**	**525**	180	900	920	
	95	70	510	**525**	**180**	900	920	
	95	70	510	180	**525**	**900**	920	
	95	70	510	180	525	**900**	**920**	
	95	70	510	180	525	**900**	920	900 no lugar
Passo 3	**95**	**70**	510	180	525	900	920	
	70	**95**	**510**	180	525	900	920	
	70	95	**510**	**180**	525	900	920	
	70	95	180	**510**	**525**	900	920	
	70	95	180	510	**525**	900	920	525 no lugar
Passo 4	**70**	**95**	180	510	525	900	920	
	70	**95**	**180**	510	525	900	920	
	70	95	**180**	**510**	525	900	920	
	70	95	180	**510**	525	900	920	510 no lugar
Passo 5	**70**	**95**	180	510	525	900	920	
	70	**95**	**180**	510	525	900	920	
	70	95	**180**	510	525	900	920	180 no lugar
Passo 6	**70**	**95**	180	510	525	900	920	
	70	**95**	180	510	525	900	920	95 no lugar

1.4.2 Método da troca e partição (*Quick Sort*)

Foi proposto por C. A. Hoare em 1962, e é considerado como o mais rápido em relação métodos apresentados anteriormente. hoare partiu da premissa de que é mais rápido classificar dois vetores de $n/2$ elementos do que um de n elementos.

Assim, este método divide o vetor a ser ordenado em três segmentos, como mostrado abaixo:

O escolha do elemento **pivô**, que divide o vetor em dois segmentos pode ser realizada de forma arbitrária, mas normalmente é escolhido o elemento central, por tornar o algoritmo mais eficiente.

Em seguida é realizada uma movimentação nos elementos, de forma que no segmento 1 fiquem os elementos menores que o pivô, e no segmento 2, os elementos maiores que o pivô. Depois é a vez de aplicar o mesmo procedimento aos segmentos 1 e 2, de forma recursiva, até que o vetor original fique totalmente ordenado. Vamos ver mais de perto

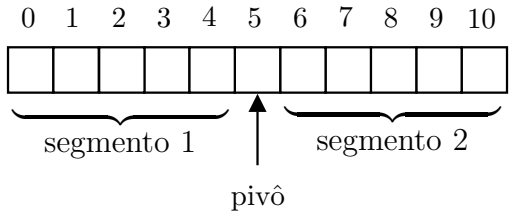


Figura 1.1: Segmentando o vetor a ser ordenado

como isto funciona através de um exemplo. Seja então o seguinte vetor, que desejamos ordenar:

	0	1	2	3	4	5	6	7	8
`vetor`	17	37	12	13	20	19	43	15	23

O procedimento *Quick Sort* pode ser então descrito da seguinte maneira:

1. Seleciona-se um elemento para ser o pivô. Pode ser qualquer um, mas geralmente escolhe-se o elemento do meio. Este é então copiado para a variável auxiliar `pivo`.

	0	1	2	3	4	5	6	7	8
`vetor`	17	37	12	13	20	19	43	15	23

`pivo` | 20 |

2. Iniciamos dois ponteiros, `esq` e `dir`, apontando para o primeiro e último elementos do vetor, respectivamente.

	0	1	2	3	4	5	6	7	8
`vetor`	17	37	12	13	20	19	43	15	23
	`esq`								`dir`

`pivo` | 20 |

3. O vetor é percorrido a partir da esquerda até que se encontre um elemento `(vetor[esq] >= pivo)`, incrementando-se o valor da variável `esq`.

	0	1	2	3	4	5	6	7	8
`vetor`	17	37	12	13	20	19	43	15	23
		`esq`							`dir`

`pivo` | 20 |

4. O vetor é percorrido a partir da direita até que se encontre um elemento `(vetor[dir] < pivo)`, decrementando-se o valor da variável `dir`.

	0	1	2	3	4	5	6	7	8
`vetor`	17	37	12	13	20	19	43	15	23
		`esq`						`dir`	

`pivo` | 20 |

5. Os elementos `vetor[esq]` e `vetor[dir]` estão fora do lugar em relação ao pivô. Desta forma, suas posições são invertidas.

	0	1	2	3	4	5	6	7	8
`vetor`	17	15	12	13	20	19	43	37	23
		`esq`						`dir`	

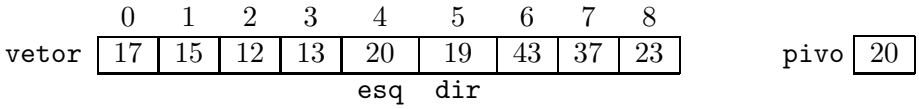
`pivo` | 20 |

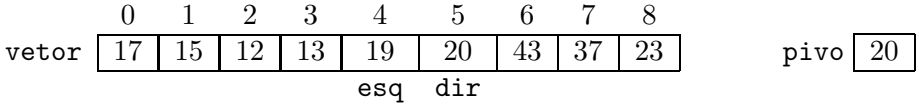
6. Incrementa-se `esq` e decrementa-se `dir`

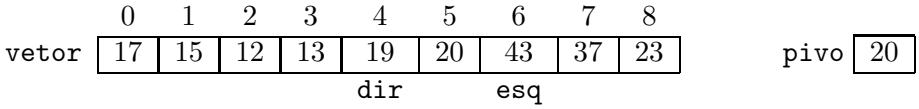
	0	1	2	3	4	5	6	7	8
`vetor`	17	15	12	13	20	19	43	37	23
			`esq`				`dir`		

`pivo` | 20 |

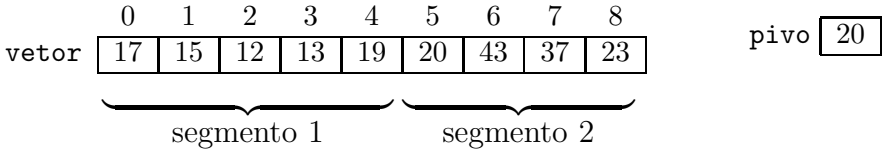
7. Este processo é repetido até que `esq` e `dir` se cruzem em algum ponto do vetor:







8. Após este passo, o pivô está na posição correta e temos dois segmentos a serem ordenados:



9. Depois, estes dois segmentos são ordenados de forma recursiva.

A seguir é mostrada uma função em C++ que implementa o algoritmo *Quick Sort* de forma recursiva. Este tem duas entradas adicionais em relação aos outros algoritmos: `i` e `j`. Estes dois parâmetros correspondem aos índices dos elementos inicial e final do vetor, respectivamente.

Mas espera um pouco: se temos um parâmetro vetorial e o tamanho do vetor na entrada, pra que precisamos ainda indicar as posições inicial e final do vetor? Isto não seria informação duplicada? A resposta é, sim e não: sim para a primeira chamada à função, quando levamos em conta o vetor inteiro. Nas chamadas recursivas (veja o passo 9), chamamos a função para processar só um pedaço do vetor, e aí sim os parâmetros `i` e `j` fazem sentido.

```
void quickSort(int vetor[], int tamanho, int i, int j)
{
  int trab,esq,dir,pivo;

  esq = i;
  dir = j;
  pivo = vetor[(int)round((esq + dir)/2.0)];
  do
  {
    while (vetor[esq] < pivo)
      esq++;
    while (vetor[dir] > pivo)
      dir--;
    if (esq <= dir)
    {
      trab = vetor[esq];
      vetor[esq] = vetor[dir];
      vetor[dir] = trab;
      esq++;
      dir--;
    }
  } while (esq <= dir);

  if (dir-i >= 0)
    quickSort(vetor,tamanho,i,dir);
  if (j-esq >= 0)
    quickSort(vetor,tamanho,esq,j);
}
```

1.5 Ordenação por Seleção

Neste caso, o processo de ordenação consiste em uma seleção sucessiva do maior (ou menor) valor contido no vetor. Feita essa escolha, coloca-se o valor maior ou menor em cada passo realizado, diretamente na sua posição correta. Esse processo é repetido para o segmento que contém os elementos ainda não selecionados. Existem dois métodos que utilizam esta abordagem: seleção direta e seleção em árvore (*heap sort*).

1.5.1 Seleção direta

Este método é bastante simples e, pelo que temos observado até agora, os algoritmos simples de entender não são muito eficientes. Este não foge à regra. Vamos a ele:

1. A cada etapa realizada, é identificado o menor elemento dentro do segmento que contém os elementos ainda não selecionados.

2. É realizada uma troca do elemento identificado na etapa anterior, com o primeiro elemento do segmento.

3. O tamanho do segmento é atualizado, ou seja, subtrai-se um do tamanho do segmento (menos um elemento).

4. Esse processo é interrompido no momento em que o segmento ficar com apenas um elemento.

Exemplo 1.3. Para entender melhor o algoritmo, nada melhor do que um exemplo. Este vai ser meio telegráfico, mas acho que dá pra entender afinal, como diz o ditado, uma figura vale mais que dez palavras:

21	27	**12**	20	37	19	17	15	TAM = 8
12	27	21	20	37	19	17	**15**	TAM = 7
12	15	21	20	37	19	**17**	27	TAM = 6
12	15	17	20	37	**19**	21	27	TAM = 5
12	15	17	19	37	**20**	21	27	TAM = 4
12	15	17	19	20	37	**21**	27	TAM = 3
12	15	17	19	20	21	37	**27**	TAM = 2
12	15	17	19	20	21	27	37	TAM = 1

O algoritmo para este método é mostrado abaixo:

```
void selecaoDireta(int vetor[], int tamanho)
{
  int menor;
  int i,j,aux;

  for (i=0;i<(tamanho-1);i++)
  {
    menor = i;
    for (j=i+1;j<tamanho;j++)
      if (vetor[j] < vetor[menor])
        menor = j;
    aux = vetor[i];
    vetor[i] = vetor[menor];
    vetor[menor] = aux;
  }
}
```

1.5.2 Heap Sort

Uma outra forma de fazer a ordenação por seleção é usar uma árvore de busca binária e o percurso em-ordem, como vimos anteriormente. Este procedimento é em geral razoavelmente rápido, mas seu desempenho varia bastante, dependendo da forma como os elementos se encontram no vetor original.

Uma forma alternativa para um algoritmo de ordenação por seleção usando uma árvore, que se mostrou bastante rápida e estável será apresentada a seguir. Esta é

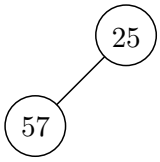
dividida em duas partes: na primeira, constrói-se uma árvore binária, e na segunda, os elementos desta árvore são retirados de forma ordenada.

Para este processo, vamos definir uma regra básica: o nó pai deve sempre apresentar o campo `info` com valor maior que o de seus filhos. Assim, à medida em que os nós são inseridos na árvore, alguns têm de trocar de posição para que esta regra seja respeitada. Vamos a um exemplo para entender melhor. Suponha que queremos ordenar os seguintes dados:

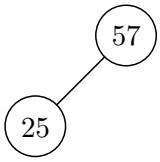
25 57 48 37 12 92 86

1. Inicialmente, colocamos o primeiro elemento (25) na raiz da árvore:

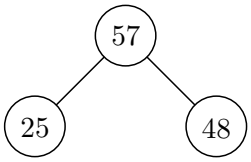
2. O segundo elemento (57) é colocado na árvore.



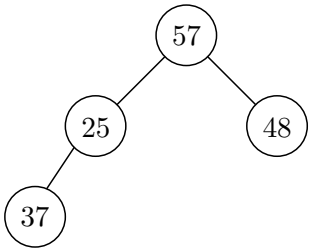
3. Seguindo a nossa regra básica, o pai deve ter o campo `info` com valor maior do que o filho. Assim, estes dois nós devem trocar de posição:



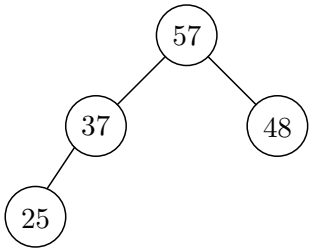
4. O terceiro elemento (48) é colocado na árvore. Veja que neste caso não há necessidade de ajuste.



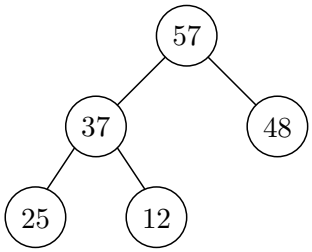
5. O quarto elemento (37) é colocado na árvore. Veja que neste caso não há necessidade de ajuste.



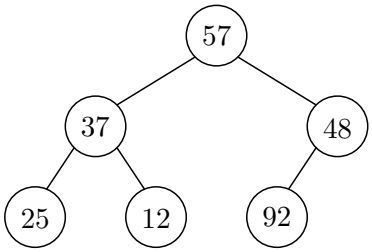
6. Este último nó inserido precisa de ajuste, pois seu valor é maior que o de seu pai. Então, ele deve trocar de posição com o pai:



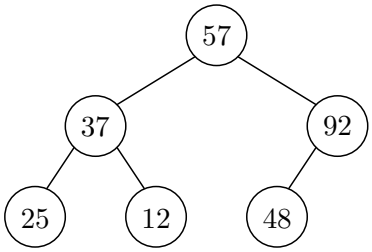
7. O quinto elemento (12) é colocado na árvore. Felizmente, também neste caso não há necessidade de ajuste.



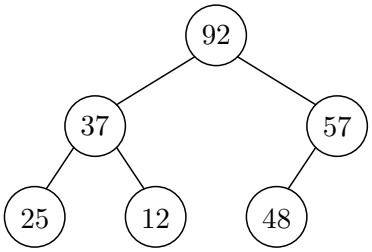
8. O sexto elemento (92) é inserido na árvore. Este vai dar trabalho ...



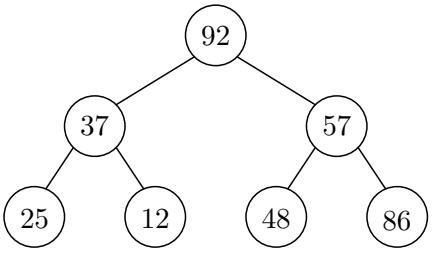
9. Primeiro, ele troca com o seu pai, pois tem valor maior:



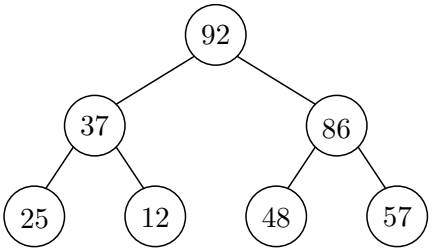
10. Ele ainda tem valor maior que o seu pai, de forma que tem que trocar de novo:



11. Finalmente, o último elemento (86) é inserido.

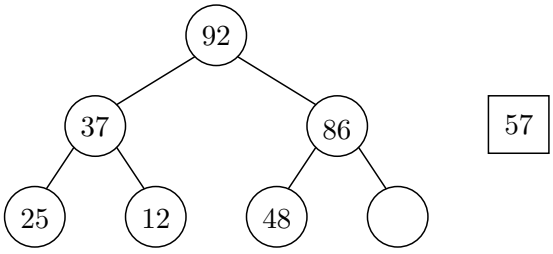


12. Como tem valor maior que o pai, troca de lugar com ele:

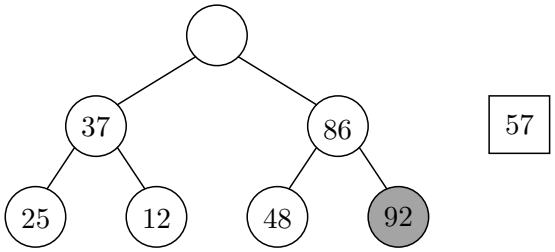


Depois desta ginástica toda, contruímos a nossa árvore, seguindo a regra básica: o nó pai tem valor maior que os seus filhos. De cara, já da pra sacar que a raiz da árvore tem o maior elemento de todos. Vamos então iniciar o processo de retirada dos elementos da árvore de forma ordenada. Para economizar memória, os elementos retirados vão ser colocados na árvore mesmo. Como? Você vai ver. O processo é o seguinte:

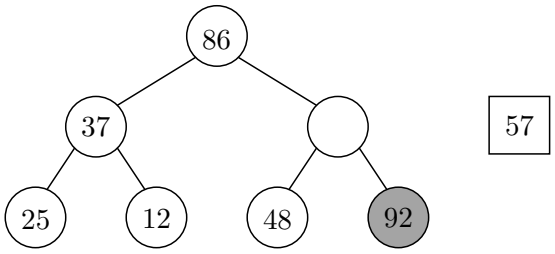
1. Primeiramente, removemos o último elemento da árvore e copiamos seu conteúdo em uma variável auxiliar externa:



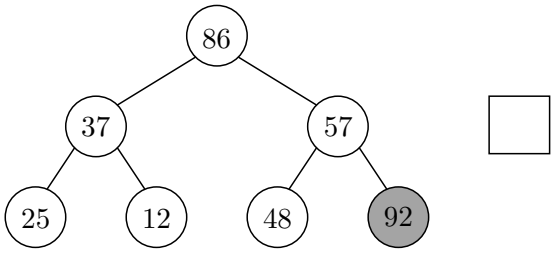
2. Depois, retiramos o nó raiz, e copiamos sua informação no nó que foi liberado no passo anterior. Observe que este nó, marcado em cinza, está fora da jogada daqui em diante.



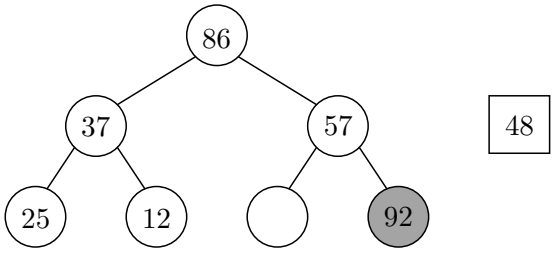
3. Agora temos que resolver quem fica com a posição da raiz: estão disputando a vaga os dois filhos e o nó que está na variável auxiliar (57). Como quem pode mais chora menos, o filho com valor 86 assume o lugar do pai:



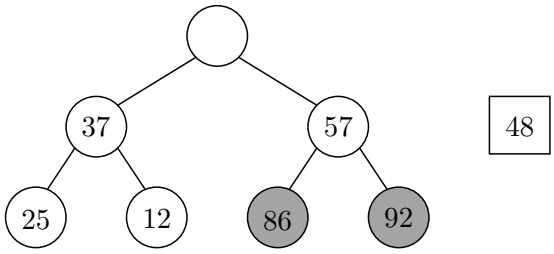
4. Quando o 86 subiu, seu lugar ficou vago. Agora disputam a vaga o seu filho esquerdo (48) e o nó na variável auxiliar (57). Pela regra, assume o posto o nó na variável auxiliar (57):



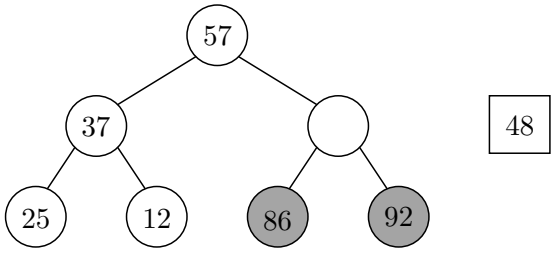
5. Repetindo o processo, agora o último nó da árvore é o (48), que vai para a variável auxiliar:



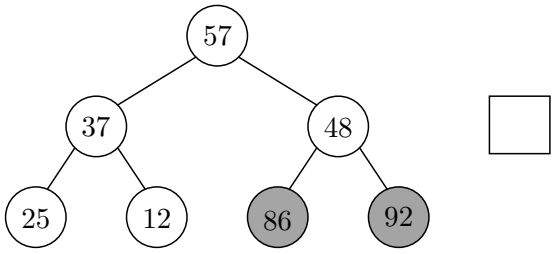
6. O nó raiz assume o lugar vazio, que deixa de fazer parte da árvore "ativa":



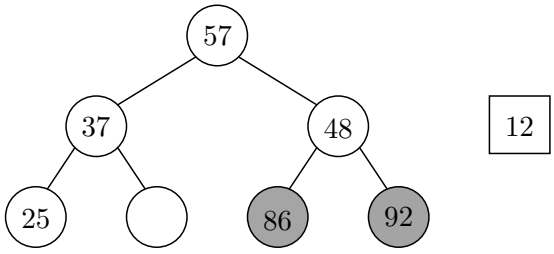
7. (57) assume o posto da raiz:



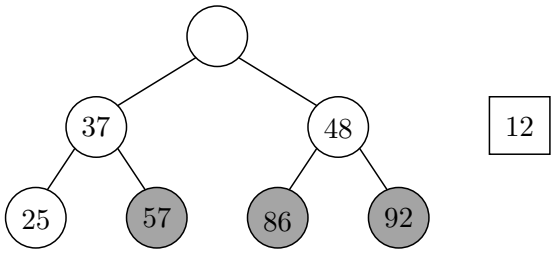
8. (48) assume o lugar deixado por (57):



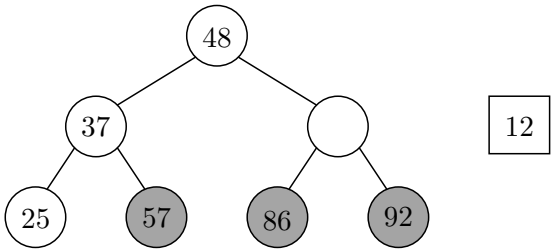
9. Repetindo este processo, teremos a seguir:



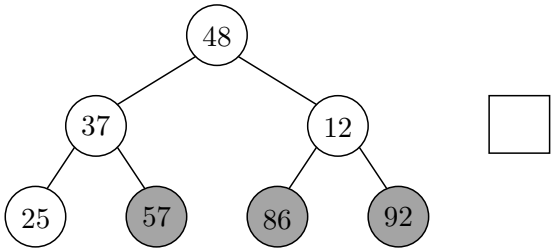
10. Depois:



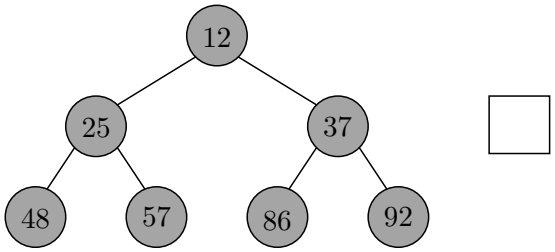
11. Reajustando a raiz:



12. E finalmente:



13. Acho que já deu pra perceber que continuando este processo, chegaremos ao resultado abaixo:



14. Um percurso em nível nesta árvore completa a ordenação. O resultado é então o vetor:

    12 25 37 48 57 86 92

Como você já deve ter percebido, este é um algoritmo bastante complexo. Vou deixar então apenas o código da função para que você tente associá-lo ao que acabei de mostrar.

```
void HeapSort(int vetor[], int tamanho)
{
  int i,elt,s,f,ivalue;

  // Fase de pre-processamento: cria heap inicial
  for (i=1;i<tamanho;i++)
  {
    elt = vetor[i];
    // pqinsert(x,i,elt)
    s = i;
    f = (s-1)/2;
    while (s > 0 && vetor[f] < elt)
    {
      vetor[s] = vetor[f];
      s = f;
      f = (s-1)/2;
    }
    vetor[s] = elt;
  }

  // Fase de selecao: remove x[0] varias vezes, insere-o em sua
  // posicao correta e acerta o heap
  for (i=tamanho-1; i>0; i--)
  {
    ivalue = vetor[i];
    vetor[i] = vetor[0];
    f = 0;

    if (i == 1)
      s = -1;
    else
      s = 1;
    if (i > 2 && vetor[2] > vetor[1])
      s = 2;
    while (s >= 0 && ivalue < vetor[s])
    {
      vetor[f] = vetor[s];
      f = s;
      s = 2*f+1;
      if (s+1 <= i-1 && vetor[s] < vetor[s+1])
        s = s+1;
      if (s > i-1)
        s = -1;
    }
    vetor[f] = ivalue;
  }
}
```

1.6 Exercícios

1. Considere a seguinte sequência de entrada:

0	1	2	3	4	5	6	7	8	9
26	34	9	0	4	89	6	15	27	44

   É solicitada uma classificação em ordem crescente sobre a sequência dada usando os algoritmos da inserção direta, *shell sort*, *bubble sort* e *quick sort*. Mostre como cada passo é executado.

2. Um vetor contém os elementos exibidos a seguir. Mostre o conteúdo do mesmo depois de ter sido executada a primeira passagem do método Shell. Considere um fator de incremento $k = 3$.

0	1	2	3	4	5	6	7	8	9	10	11	12
24	4	8	14	90	8	67	27	45	19	91	99	58

3. No vetor apresentado a seguir, os dois primeiros elementos já foram classificados usando o método de inserção direta. Qual será o valor dos outros elementos no vetor, depois de realizar mais três passagens deste algoritmo?

0	1	2	3	4	5	6	7
4	14	8	27	45	24	99	58

4. No vetor apresentado a seguir, os dois primeiros elementos já foram classificados usando o método da bolha. Qual será o valor dos outros elementos no vetor, depois de realizar mais três passagens deste algoritmo?

0	1	2	3	4	5	6	7
4	14	8	27	45	24	58	99

5. Considere o vetor abaixo. Usando o *quick sort*, mostre o conteúdo do vetor depois que o primeiro pivô tiver sido colocado na posição correta. Identifique as duas sublistas existentes neste momento.

0	1	2	3	4	5	6	7	8	9	10
45	79	23	8	99	57	35	3	39	36	46

6. Suponha o seguinte vetor:

0	1	2	3	4	5
48	4	22	33	57	93

   Depois de duas passagens de um algoritmo de ordenação, o seu conteúdo passou a ser:

0	1	2	3	4	5
4	22	48	33	57	93

   Qual foi o algoritmo de ordenação utilizado (seleção, bolha ou inserção direta)? Justifique sua resposta.

7. Repita o exercício anterior para os vetores abaixo:

   Antes:

0	1	2	3	4	5
81	73	67	45	22	34

   Depois

0	1	2	3	4	5
22	34	67	45	81	73

8. Idem para os vetores:

   Antes:

0	1	2	3	4	5
48	4	67	33	57	93

   Depois

0	1	2	3	4	5
4	48	67	33	57	93

9. Suponha o seguinte conjunto numérico: 26, 20 28, 54, 38, 22, 46. Coloque-o em ordem utilizando o método *heap sort*, descrevendo passo a passo a evolução do processo de ordenação.

10. Os exercícios de ordenação apresentados até agora solicitam o desenvolvimento de uma ordenação que pode ser classificada como destrutiva, porque o vetor original é destruído e substituído por sua versão ordenada. Uma boa alternativa é criar um vetor auxiliar cujos índices representam a posição dos elementos no vetor a ser ordenado. Faça um programa em C++ que use o vetor auxiliar para implementar o método de ordenação por seleção direta.

11. Reimplemente os algoritmos desta seção para ordenar listas ligadas ao invés de vetores.

12. Modifique a função que implementa o algoritmo *bubble sort*, de modo que este retorne o número de trocas realizadas até a ordenação final do vetor.

Capítulo 2

Pesquisa

A **pesquisa** consiste na verificação da existência de um valor dentro de um vetor. Trocando em miúdos, pesquisar um vetor consiste em procurar dentre suas componentes qual(is) possui(em) um determinado valor. Nesta seção iremos estudar dois métodos de pesquisa bastante difundidos: a **pesquisa sequencial** e **pesquisa binária**.

2.1 Método de pesquisa sequencial.

A pesquisa **sequencial** ou **linear** é o método mais objetivo para se encontrar um elemento particular num vetor não classificado, isto é, cujos elementos não estão ordenados segundo algum critério.

Esta técnica envolve a simples verificação de cada componente do vetor sequencialmente (uma após a outra) até que o elemento desejado seja encontrado (neste caso diz-se que a pesquisa foi **bem sucedida**) ou que todos os elementos do vetor tenham sido verificados sem que o elemento procurado tenha sido encontrado (pesquisa **malsucedida**).

Abaixo tem-se uma função que implementa a pesquisa sequencial. Ela retorna o índice em que o elemento `x` foi encontrado em `vetor`, ou -1, se a pesquisa for malsucedida.

```
int sequencial(int vetor[],int tamanho, int x)
{
  bool achou = false; // var aux p/ busca
  int i=0; // contador

  while (achou==false && i<tamanho)
    achou = vetor[i++]==x;

  if (achou)
    return (i-1);
  else
    return -1;
}
```

2.2 Método de pesquisa binária.

Quando os elementos de um vetor estão previamente classificados segundo algum critério, então pesquisas muito mais eficientes podem ser conduzidas. Entre elas destaca-se o método de **pesquisa binária**.

Seu funcionamento é o seguinte: inicialmente o vetor é classificado (por exemplo, em ordem crescente). O elemento que divide o vetor ao meio (ao menos aproximadamente) com relação ao seu número de componentes é localizado e comparado ao valor procurado. Se ele for igual ao valor procurado a pesquisa é dita bem sucedida e é interrompida. No caso dele ser maior que o valor procurado, repete-se o processo na primeira metade do vetor. No caso do elemento central do vetor ser menor que o valor procurado, repete-se o processo na segunda metade do vetor. Este processo continua até que o elemento desejado seja localizado (pesquisa bem-sucedida), ou então, até que não reste mais um trecho do vetor a ser pesquisado (pesquisa mal-sucedida).

Este método tem a desvantagem de exigir que o vetor seja previamente ordenado para seu correto funcionamento, o que não acontece no caso da pesquisa sequencial. Por outro lado, é **em média** muito mais rápido que o método de pesquisa sequencial.

A seguir é apresentada uma função que implementa o método de pesquisa binária para um vetor do tipo `int`. As saídas dela são as mesmas da função anterior (pesquisa sequencial).

```
int binaria(int vetor[],int tamanho, int x)
{
  bool achou; // var aux p/ busca
  int baixo, meio, alto; // var aux

  baixo = 0;
  alto = tamanho-1;
  achou = false;
  while ((baixo <= alto) && (achou == false))
  {
    meio = (baixo + alto) / 2;
    if (x < vetor[meio])
      alto = meio - 1;
    else
      if (x > vetor[meio])
        baixo = meio + 1;
      else
        achou = true;
  }
  if (achou)
    return meio;
  else
    return -1;
}
```

2.3 Exercícios

1. A eficiência da busca sequencial pode ser aumentada se colocarmos nas posições iniciais os elementos que ocorrem com maior frequência. Para isto, duas estratégias são utilizadas:

   (a) **mover-para-frente**: sempre que uma pesquisa tiver êxito, o elemento encontrado é movido para o início da lista.

   (b) **transposição**: sempre que uma pesquisa tiver êxito, o elemento encontrado troca de lugar com o seu antecessor.

   Escreva funções que implementem estas duas estratégias.

2. Reimplemente os algoritmos de busca sequencial e busca binária para listas ligadas.

3. Reimplemente os algoritmos de busca sequencial e busca binária para listas circularmente ligadas.

4. Suponha que uma tabela ordenada esteja armazenada como uma lista circular com dois ponteiros externos: `vetor` e `outro`, que seguem as seguintes regras:

   - `vetor` aponta sempre para o nó com o menor elemento
   - `outro` aponta inicialmente para o mesmo nó que `vetor`, mas depois aponta para o último elemento encontrado
   - se uma busca for mal-sucedida, `outro` aponta novamente para o mesmo nó que `vetor`

   Faça uma função em C++ que implemente este método e retorne um ponteiro para o nó recuperado, ou um ponteiro nulo, se a busca for mal-sucedida.

   Se a tabela está ordenada, explique como o ponteiro `outro` ajuda a reduzir o número de comparações numa busca.

5. Imagine um programador que escreva o seguinte código:

```
if (c1)
  if (c2)
    if (c3)
      ...
        if (cn)
          comando;
```

   onde c_i é uma condição verdadeira ou falsa. Observe que a reordenação das condições numa sequência diferente resulta em um programa equivalente porque `comando` só será executado se todos os c_i forem verdadeiros. Suponha que $time(i)$ seja o tempo necessário para avaliar a condição c_i e $prob(i)$ seja a probabilidade da condição c_i ser verdadeira. Em que ordem as condições devem ser organizadas para tornar o programa mais eficiente?

6. Seja um vetor ordenado, e o seguinte método de pesquisa:

   - suponha que uma variável `p` armazena o índice do úlimo elemento encontrado
   - a busca começa sempre na posição indicada por `p`, mas pode prosseguir em ambas as direções

   Escreva uma função que implemente este método de busca. A sua função deve ter o seguinte cabeçalho:

   ```
   int busca(int vetor[],int tamanho,int p,int x)
   ```

   No caso de uma busca bem sucedida, a função deve retornar a posição em que o elemento `x` foi encontrado. Caso contrário, ela deve retornar o valor `-1`.

7. Modifique o algoritmo de busca binária de modo que, no caso de uma busca mal-sucedida, ele retorne o índice i tal que o elemento procurado esteja entre i e $i+1$. Se o elemento procurado for menor que o menor elemento da lista, o algoritmo deve retornar -1, e se for maior que o maior elemento da lista, o algoritmo deve retornar n. Observe que a lista tem n elementos: $[0, n-1]$.

8. Suponha que temos o seguinte vetor ordenado:

   1 2 3 4 5 6 7

   Se estivermos procurando pelo elemento 4, quantas comparações irão ser feitas pelo algoritmo de busca sequencial? E pelo algoritmo de busca binária? Justifique suas respostas.

9. Faça uma função que implemente o algoritmo de busca sequencial, partindo do final do vetor ao invés do início.

Capítulo 3

Listas Ligadas

Estruturas de dados são formas de armazenar os dados. Estas podem ser:

- **Variáveis simples**: armazenam apenas um dado de cada vez. Também só armazenam dados de um único tipo (`int`, `float`, `char`, etc.).
- **Estruturas**: armazenam dados de vários tipos em uma estrutura única.
- **Vetores e Matrizes**: armazenam vários dados de um mesmo tipo (inclusive estruturas), mas precisamos conhecer de antemão a quantidade máxima de dados armazenados.
- **Listas ligadas**: podem ser vistas como vetores de tamanho variável, ou seja, vão crescendo e diminuindo de acordo com as necessidades.

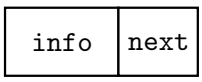
Nosso objetivo neste capítulo é estudar as listas ligadas, sua estrutura e utilidade, bem como os algoritmos para manipulá-las. Vamos começar então.

3.1 Estrutura das Listas Ligadas

As listas ligadas são formadas por **nós**. Cada nó é constituído de duas partes, chamadas de **campos**. Os campos são:

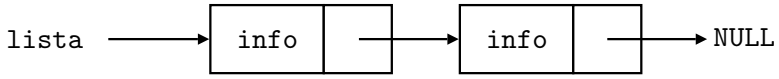
- **info**: armazena os dados da lista.
- **next**: contém um ponteiro que aponta para o próximo nó da lista.

De forma gráfica, temos:

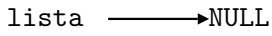


A lista ligada inteira é acessada a partir de um ponteiro externo `lista`, que aponta para (contém o endereço de) o primeiro nó da lista.

O último nó da lista tem o seu campo de endereço para o próximo nó preenchido com o valor NULL, que não é um endereço válido, e serve apenas para indicar o final de uma lista, como mostrado abaixo:



A lista sem nós é chamada de **lista vazia** ou **lista nula**. O valor do ponteiro externo lista neste caso é `NULL`, indicando que não há nós:



3.1.1 Declaração de um nó

Em C++, podemos declarar um nó usando uma `struct`, como mostrado abaixo, para uma lista ligada que armazena somente números inteiros:

```
struct no
{
  int info;
  struct no *next;
}
```

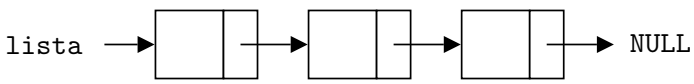
Muitos autores consideram uma boa prática de programação definir variáveis do tipo ponteiro. Desta forma, um nó seria declarado da seguinte forma:

```
typedef no* noptr; // define um tipo ponteiro
noptr p;           // declara um ponteiro p para um nó
```

Agora que já sabemos como declarar um nó, vamos aprender como criar nós e começar a brincar com eles. Primeiramente vamos aprender como inserir um nó no início de uma lista ligada.

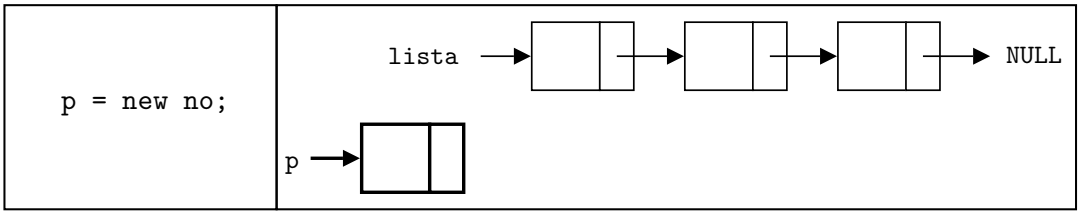
3.1.2 Inserindo um nó no início da lista

Vamos supor inicialmente que a lista já contenha alguns nós, como mostrado abaixo:

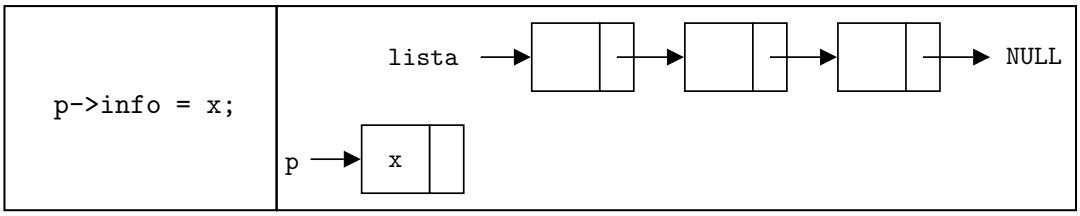


Para inserir um novo nó no início desta lista, devemos seguir os seguintes passos:

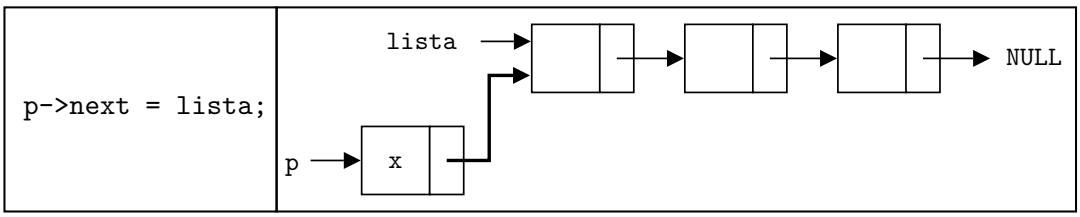
1. alocar memória para um novo nó auxiliar `p`:



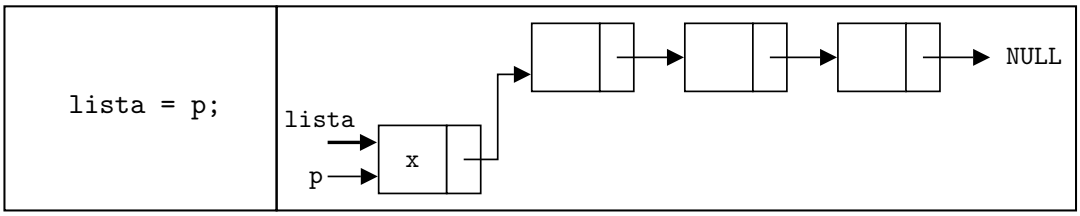
2. colocar a informação no campo `info` deste nó:



3. o endereço do primeiro nó da lista é colocado no campo `next` do nó recém-criado:



4. o ponteiro externo `lista` é atualizado, e passa a apontar para o nó recém criado:



5. depois disso, o ponteiro auxiliar `p` pode ser usado para outra operação.

Os passos acima podem ser encapsulados em uma função. Em C++ esta ficaria da seguinte forma:

```
void push(noptr &lista,int x)
{
  no *p;
  p = new no;
  p->info = x;
  if (lista == NULL) // verifica se é o primeiro nó a ser inserido
    p->next = NULL;
  else // já existem nós na lista
    p->next = lista;
  lista = p;
}
```

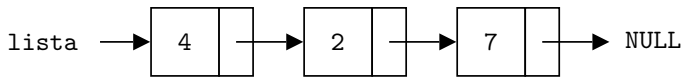
Na função acima, `lista` é um ponteiro que aponta para o início da lista ligada, e `x` contém a informação a ser colocada no campo `info` do nó recém inserido.

Você pode achar estranho o nome `push` dado à função, mas isto ficará claro mais adiante, quando estudarmos estruturas chamadas de pilhas. Por enquanto, aceite o nome como uma excentricidade nossa.

Agora, já que aprendemos a inserir nós no início da lista, vamos dar uma olhada em como remover nós do início da lista.

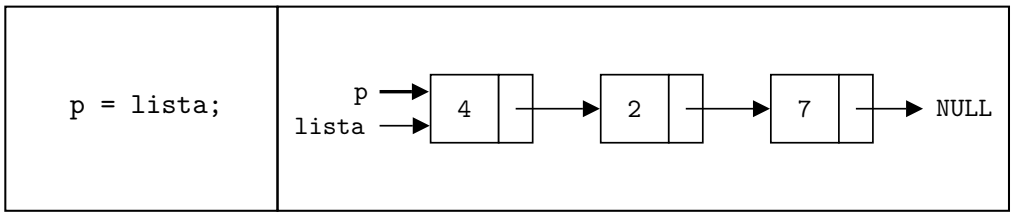
3.1.3 Removendo um nó do início da lista

Vamos supor inicialmente que a lista contenha alguns nós, como mostrado abaixo:

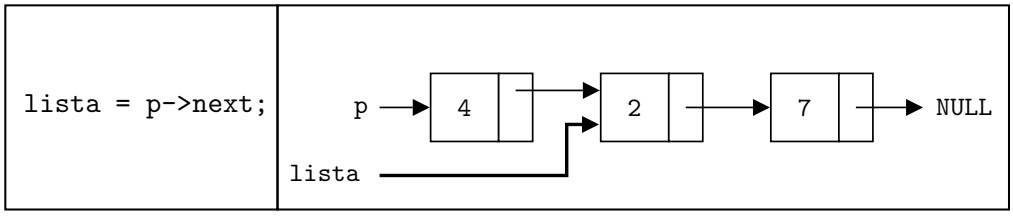


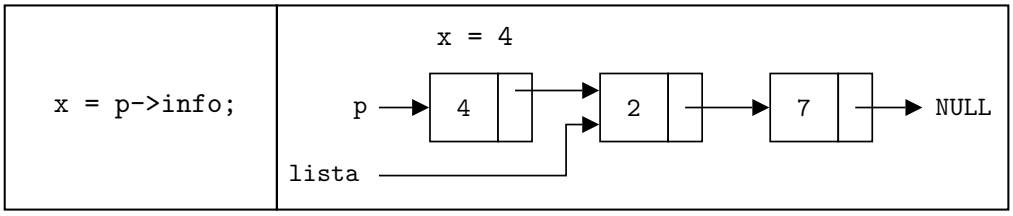
Para remover um nó do início da lista, devemos seguir o seguinte procedimento:

1. fazer um ponteiro auxiliar `p` apontar para o primeiro nó da lista.



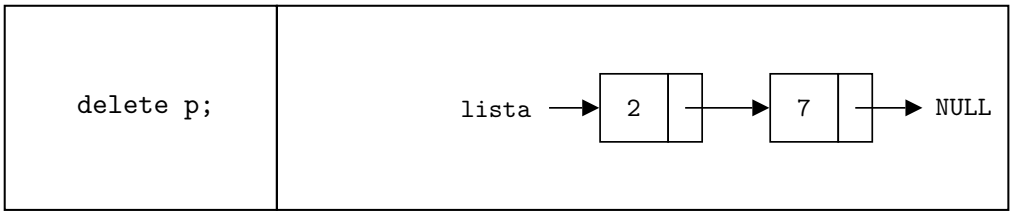
2. `lista` aponta agora para o sucessor de `p` (que passa a ser o primeiro nó)





3. recuperamos a informação do primeiro nó da lista (que deve ser removido), e a armazenamos em uma variável `x` separada.

4. finalmente, desalocamos o nó inicial da lista (apontado por `p`).



Este procedimento está encapsulado na função `pop()`, mostrada abaixo:

```
int pop(noptr &lista, int &x)
{
  no *p;

  if (lista == NULL) // lista vazia, não há o que remover
    return -1;

  else // remoção do primeiro nó da lista
  {
    p = lista;
    lista = p->next;
    x = p->info;
    delete p;
return 0;
  }
}
```

Esta função retorna -1 se a lista já estiver vazia, pois neste caso não há o que remover. Se existir pelo menos um nó na lista, ela vai retornar 0 (remoção bem sucedida), e o valor de `x` é retornado como um parâmetro passado por referência à função.

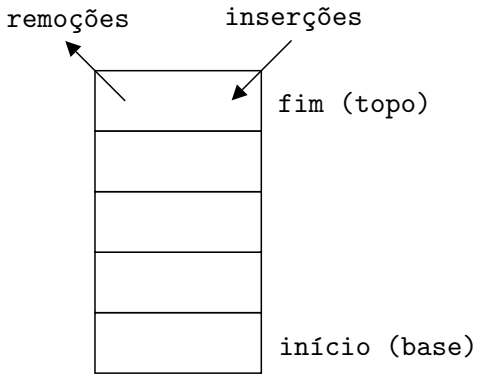
3.2 Listas lineares com disciplinas de acesso

Disciplinas de acesso referem-se à forma como são acessados os elementos de uma lista. Verificam se as inserções, as remoções e os acessos de consulta são realizados no primeiro ou no último elemento das listas. Elas podem ser classificadas em pilhas, filas, ou, ainda, em filas duplas. Vamos ver agora, em detalhes, cada uma destas estruturas.

3.2.1 Pilhas

"Os últimos serão os primeiros"

A pilha é uma lista linear na qual todos os acessos (inserção, remoção ou consulta) em uma só extremidade, denominada **topo**, como mostrado na figura abaixo:



Analisando-se a possibilidade de inserir e remover elementos em uma pilha, podemos perceber claramente um padrão de comportamento que indica que o **último** elemento a ser inserido (empilhado), é o **primeiro** elemento a ser removido (desempilhado). Você pode pensar nesta estrutura como uma pilha de pratos: sempre colocamos um novo prato no topo e, se precisamos retirar um prato, o fazemos a partir do topo também (pelo menos se você não quiser causar um transtorno na cozinha). Este é o motivo das pilhas serem também referenciadas como uma estrutura **LIFO** (Last In, First Out).

Operações sobre as pilhas

Existem duas operações básicas que podemos executar sobre as pilhas:

- **empilhar** um novo elemento e,
- **desempilhar** o elemento do topo.

A forma mais usual (e eficiente) de implementação de uma pilha é através do uso de listas ligadas, que acabamos de aprender. Neste caso, uma pilha vazia consiste de um nó apontando para o valor `NULL`, como vimos anteriormente.

Para empilhar um elemento no topo da pilha, usamos a função `push`, e para retirar o elemento do topo da pilha, usamos a função `pop`. Aliás, estes nomes foram dados por causa das pilhas.

Abaixo, temos um exemplo bem simples, usando as funções `push` e `pop` para ilustrar o funcionamento básico de uma pilha. Para simplificar as coisas, vamos supor que a parte info de cada nó contém apenas um número inteiro como vínhamos fazendo até agora. As funções push e pop são as mesmas mostradas acima, e portanto não vão ser repetidas. Vou apenas mostrar a função `main()`:

```
int main()
{
  noptr pilha = NULL;
  int x;
  // Colocando elementos na pilha
  cout << "Entre com os números a armazenar (-1 para terminar)\n";
  cin >> x;
  while (x != -1)
  {
    push(pilha,x);
    cout << "Entre com os números a armazenar (-1 para terminar)\n";
    cin >> x;
  }
  // Retirando elementos da pilha
  cout << "Elementos armazenados:\n";
  while (pilha != NULL)
  {
    pop(pilha,x);
    cout << x << endl;
  }
  return 0;
}
```

Neste programa, pede-se ao usuário que entre com números inteiros. Cada número digitado é então empilhado, até seja digitado o valor -1. Neste momento, o programa vai desempilhando os dados, até esvaziar a pilha.

3.2.2 Filas

As filas são um pouco mais democráticas do que as pilhas. Nestas estruturas, as inserções são realizadas no final, e as remoções no início. Pense em uma fila de supermercado: o primeiro a chegar é o primeiro a ser atendido (furar fila não vale). Por esta razão, estas estruturas são também conhecidas como **FIFO** (First In, First Out).

Representação

Nas pilhas, todas as inserções e remoções são feitas apenas em uma das extremidades da lista ligada (o topo). Já nas filas, temos operações em ambas as extremidades. Desta forma, precisamos ter dois ponteiros ao invés de um: o primeiro apontando para o primeiro nó (início da fila) e o outro apontando para o último nó (final da fila). Em C++ podemos usar uma `struct` para fazer isso, da seguinte maneira:

```
struct fila
{
  noptr inicio;
  noptr fim;
};
```

Assim, uma fila `q` pode ser declarada da seguinte maneira em um programa C++:

```
fila q;

q.inicio = NULL;
q.fim = NULL;
```

O nó inicial é então `q.inicio`, e o nó final, `q.fim`. Para uma fila inicialmente vazia, devemos ter os dois ponteiros com o valor `NULL` (não estão apontando para nenhum nó ainda).

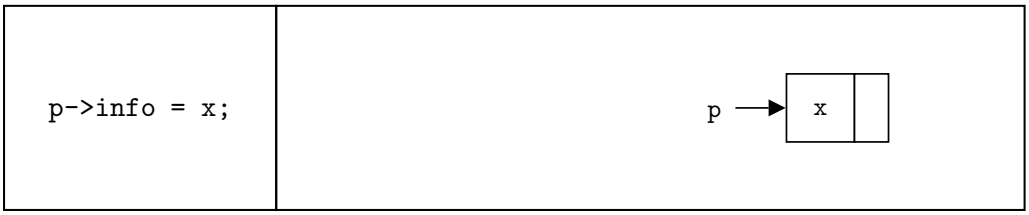
Agora que já sabemos o básico, vamos aprender como inserir e remover nós de uma fila. Mãos à obra!

Inserindo um nó no final da fila

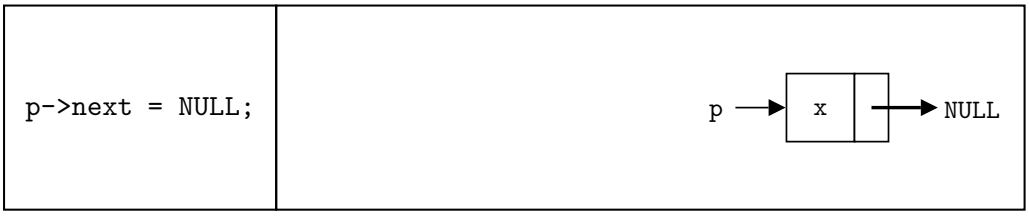
Para inserir um nó na fila, devemos fazê-lo no fim (lembre-se: se chegamos por último, devemos aguardar nosso lugar na fila). Isto pode ser feito seguindo o seguinte procedimento:

1. alocar memória para um novo nó auxiliar `p`:

2. colocar a informação no campo info deste nó:

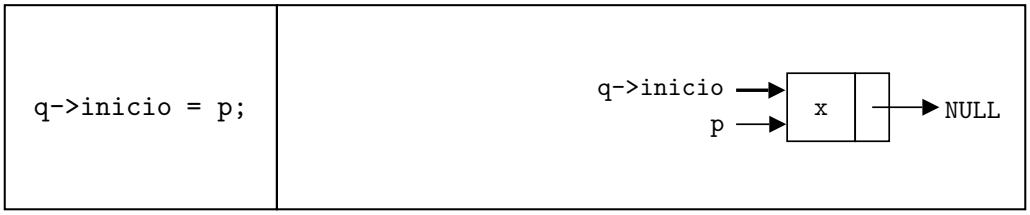


3. o campo `next` do nó recém criado deve apontar para `NULL`, pois este é o último nó da fila.

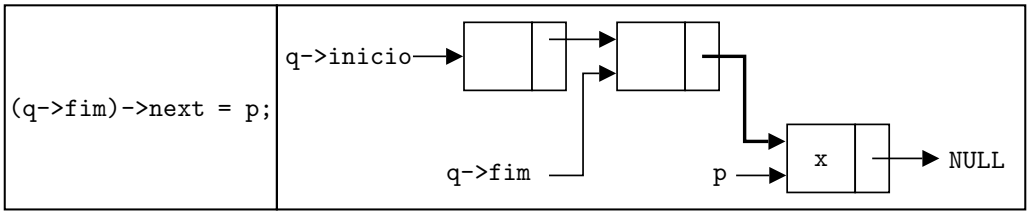


4. agora temos duas situações possíveis:

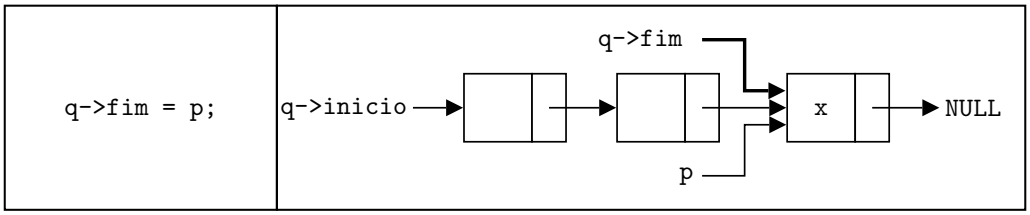
   (a) se a fila está inicialmente vazia, então o nó recém criado é também o primeiro nó, e desta forma, `q.inicio` deve apontar para este nó:



(b) se já existem outros nós na lista, o campo `next` do último nó deve apontar para `p`.



5. finalmente, devemos atualizar o ponteiro `q->fim`, que deve apontar para o nó recém criado:



Todo este procedimento pode ser encapsulado em uma função para facilitar o seu uso em programas mais complexos. Vamos chamar esta função de `insere`. O código em C++ dela se encontra abaixo. Esta função tem como parâmetros um ponteiro para uma struct `fila` e o dado a ser inserido.

```
void insere (fila &pq, int x)
{
  noptr p;
  p = new no;
  p->info = x;
  p->next = NULL;
  if (pq->fim == NULL) // A fila não tem nenhum nó ainda
    pq->inicio = p;
  else                 // A fila já tem pelo menos um nó
    (pq->fim)->next = p;
  pq->fim = p;
}
```

Removendo um nó no início da fila

A remoção de um nó em uma fila deve ser realizada no início da mesma.

```
int remove(fila &pq,int &x)
{
  noptr p;
  // Verificando se a fila já está vazia
  if (pq->inicio == NULL)
    return -1;

  // Removendo nó do início da fila
  p = pq->inicio;
  x = p->info;
  pq->inicio = p->next;
  // Para fila vazia, os dois ponteiro apontam para NULL
  if (pq->inicio == NULL)
    pq->fim = NULL;
  delete p;
  return 0;
}
```

Esta função tem dois parâmetros: um ponteiro para a fila, e o dado a ser recuperado, passado por referência. Para ilustrar o funcionamento das funções insere e remove temos abaixo um exemplo de programa que as utiliza. Estude-o com cuidado e, de preferência, implemente-o e rode-o passo a passo para entender bem o que ele está fazendo.

```
int main()
{
  fila q;
  int x;
  // Inicializando a fila
  q.inicio = NULL;
  q.fim = NULL;
  // Inserindo elementos na fila
  cout << "Entre com os elementos a armazenar (-1 para terminar).\n";
  cin >> x;
  while (x != -1)
  {
    insere(q,x);
    cout << "Entre com os elementos a armazenar (-1 para terminar).\n";
    cin >> x;
  }
  // Removendo elementos da fila
  cout << endl << "Elementos armazenados: " << endl;
  while (q.fim != NULL)
  {
    remove(q,x);
    cout << x << endl;
  }
  return 0;
}
```

3.3 Listas ligadas como estruturas de dados

As listas ligadas não servem apenas para implementar pilhas e filas, mas também para outras aplicações muito úteis. Antes de mostrá-las entretanto, precisamos aprender mais alguns truques com as listas ligadas.

Removendo um nó depois de outro

A primeira coisa que vamos aprender é como remover um nó qualquer do meio de uma lista. Como em uma lista ligada só podemos acessar um nó a partir do seu predecessor, só podemos remover um nó que vem depois de outro (lembre-se que um nó só possui um ponteiro para o próximo nó, e não para o anterior). A função `delafter` mostrada abaixo, dá conta do recado:

```
int delafter(noptr &p, int &x)
{
  noptr q;

  if(p->next == NULL) // não existe nó depois de p
    return -1;
  else // removendo nó que vem depois de p
  {
    q = p->next;
    x = q->info;
    p->next = q->next;
    delete q;
    return 0;
  }
}
```

Se não houver nó depois de `p` para ser removido, esta função retorna -1; caso contrário, ela remove o nó que vem depois daquele apontado por `p`, e retorna 0. A informação do nó removido é retornada na variável `x`. Faça um desenho e acompanhe a execução desta função passo a passo para entendê-la bem.

Removendo todos os nós com uma dada informação

Com esta função, já podemos fazer uma aplicação bastante útil: remover todos os nós que contenham uma dada informação. Por exemplo, suponha que você tenha em uma lista ligada as informações de seus clientes devedores. Um dos campos da parte `info` poderia ser por exemplo uma variável lógica que assume o valor `true` se o cara está te devendo uma grana. Se ele pagou a dívida, então o nó referente a ele deve ser removido da lista.

Para simplificar as coisas, vamos continuar usando os nós com a estrutura simples que estamos acostumados. Deixo a extrapolação para um caso real como exercício. Então, vamos fazer uma função que varra uma lista ligada e remova todos os nós que contenham uma dada informação `x` em seu campo info. Mas, como já vimos, para eliminar um nó de uma lista, seu predecessor deve ser conhecido. Por essa razão,

vamos usar dois ponteiros `vaca` e `boi`: `vaca` é usado para atravessar a lista, e `boi` aponta sempre para o predecessor de `vaca`. A função usa a operação `pop` para remover nós do início da lista, e a operação `delafter` para remover nós do meio da lista. O código é mostrado abaixo:

```
void removetudo(noptr &lista, int x)
{
  noptr vaca,boi;
  int aux;

  boi = NULL;
  vaca = lista;

  while (vaca != NULL)
  {
    if (vaca->info == x)
      if (boi == NULL) // remove o primeiro no da lista
      {
        pop(lista,aux);
        vaca = lista;
      }
      else
      {
        vaca = vaca->next;
        delafter(boi,aux);
      }
    else // continua atravessando a lista
    {
      boi = vaca;
      vaca = vaca->next;
    }
  }
}
```

A prática de usar dois ponteiros, um depois do outro, é muito comum ao trabalhar com listas ligadas. Vou deixar como exercício a elaboração de um programa principal que crie uma lista ligada e insira alguns nós na mesma (você pode usar a função `push` para fazer isso). Depois o programa deve pedir ao usuário que digite o valor que os nós a serem eliminados deve conter, e chamar a função `removetudo` para fazer o trabalho sujo.

Iserindo um nó depois de outro

Vamos agora aprender um outro truque, que vai ser necessário para a nossa próxima aplicação: inserir um nó no meio da lista. Como você deve lembrar, usamos `push` para inserir um nó no início da lista e `insere` para inserir um nó no final.

Um item só pode ser inserido depois de um determinado nó, não antes, pois não há como saber que é o predecessor de um nó (lembre que um nó só tem um ponteiro para

o próximo nó, e não para o anterior). Com isso, a função para inserir um nó depois do nó apontado por `p` é mostrada abaixo:

```
void insertafter(noptr &p, int x)
{
  noptr q;

  q = new no;
  q->info = x;
  q->next = p->next;
  p->next = q;
}
```

Criando uma lista ordenada

Com isso, já temos condições de implementar a nossa segunda aplicação: a criação de uma lista ordenada, onde os itens menores precedam os maiores. O objetivo é inserir um novo nó em seu local correto, de modo a manter a lista ordenada. Isto pode ser feito varrendo-se a lista até que um elemento maior que o elemento a ser inserido seja encontrado, e então inserir o novo nó na posição adequada. Para isso, o algoritmo usa a função `push` para incluir um nó no início da lista, e a função `insafter` para incluir um nó no meio da lista. Eis o código desta função:

```
void place(noptr &lista, int x)
{
  noptr p,q;

  // Verificando posição em que o nó deve ser inserido
  q = NULL;
  for (p=lista; p!=NULL && x > p->info; p = p->next)
    q = p;

  // Neste ponto, um no contendo x deve ser inserido
  if (q == NULL) // insere x no início da lista
    push(lista,x);
  else           // insere x no meio da lista
    insertafter(q,x);
}
```

3.4 Filas de prioridade

Tanto a pilha como a fila são estruturas de dados cujos elementos estão ordenados com base na seqüência na qual foram inseridos. Se existir uma ordem intrínseca entre os próprios elementos (por exemplo, ordem numérica ou alfabética), ela será ignorada.

A fila de prioridade é uma estrutura de dados na qual a classificação intrínseca dos elementos determina os resultados de suas operações básicas. Existem dois tipos de filas de prioridade:

- **ascendente**: os itens podem ser inseridos de forma arbitrária, mas apenas o **menor** deles pode ser removido.
- **descendente**: os itens podem ser inseridos de forma arbitrária, mas apenas o **maior** deles pode ser removido.

Os elementos de uma fila de prioridade não precisam ser números ou caracteres que possam ser comparados diretamente. Eles podem ser estruturas complexas, classificadas por um ou vários campos. Por exemplo, as listagens do catálogo telefônico consistem de nomes, sobrenomes, endereços e números de telefone, e são (geralmente) classificados pelo sobrenome.

Ocasionalmente, o campo pelo qual os elementos de uma fila de prioridade são classificados não faz sequer parte dos próprios elementos; ele pode ser um valor externo, especial, usado de forma específica para o propósito de classificar uma fila de prioridade. Por exemplo, podemos usar a hora de inserção para fazer a classificação.

Implementação

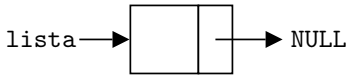
Podemos implementar estas estruturas usando listas ligadas da seguinte maneira:

- **Fila de prioridade ascendente**: podemos usar a função `place`, que definimos anteriormente, para inserir os elementos e manter a lista ordenada, e a função `pop` para remover o menor elemento.
- **Fila de prioridade descendente**: usamos `place` para inserir os elementos na lista e mantê-la ordenada. Como o maior elemento é o último, usamos `remove` para remover os elementos.

3.5 Nós de cabeçalho

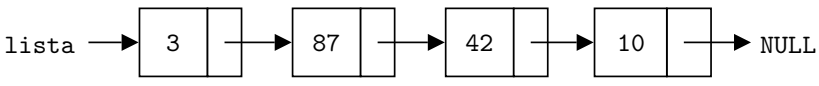
Opcionalmente, é desejável manter um nó adicional no início de uma lista. Este nó não representa um item na lista e é chamado de **nó de cabeçalho** ou **cabeçalho de lista**.

Com essa representação, uma lista vazia não seria mais representada por um ponteiro nulo, mas por uma lista com um único nó de cabeçalho, como mostrado abaixo:



Mas, se este nó não armazena um elemento da lista, então pra que serve? Geralmente, a parte `info` deste nó é usada para manter informações globais sobre a lista. Vejamos alguns exemplos:

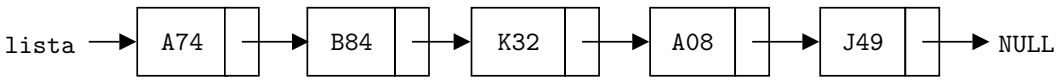
1. A figura abaixo mostra uma lista na qual a parte `info` do nó de cabeçalho contém o número de nós na lista (obviamente, sem incluir o nó de cabeçalho):



   Nessa estrutura de dados, são necessárias mais etapas para incluir ou eliminar um item da lista porque a contagem do nó de cabeçalho precisa ser acertada.

Entretanto, o número de itens na lista pode ser obtido diretamente a partir do nó de cabeçalho, sem atravessar a lista inteira.

2. Outro exemplo do uso de nós de cabeçalho é o seguinte: suponha que uma fábrica monte máquinas a partir de unidades menores. Uma determinada máquina (número de inventário A74) poderia ser formada por uma variedade de partes diferentes (números B84, K32, A08, J49). Essa montagem poderia ser representada por uma lista ligada, como mostra a figura abaixo, em que cada item representa um componente, e o nó de cabeçalho representa a montagem inteira:



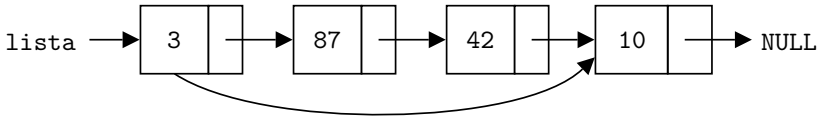
Evidentemente, os algoritmos para operações como `empty`, `push`, `pop`, `insere` e `remove` precisam ser reescritos de modo a considerar a presença de um nó de cabeçalho. A maioria das rotinas fica um pouco mais complexa, mas algumas, como `insere`, tornam-se mais simples, uma vez que um ponteiro de lista externa nunca é nulo. Deixarei a readaptação das rotinas como exercício para você treinar um pouco. Parece difícil, mas depois que a gente pega o jeito da coisa, dá pra ver que o bicho não é tão feio.

As rotinas `insafter` e `delafter` não precisam ser alteradas. Na realidade, quando usamos nós de cabeçalho, usamos estas funções ao invés de `push` e `pop`, pois o primeiro item passa a ser o segundo nó da lista, e não o primeiro.

Outras possibilidades

Se a parte `info` de um nó pode conter um ponteiro, surgem novas possibilidades para o uso de um nó de cabeçalho. Vamos ver então duas possíveis aplicações:

1. A parte `info` de um nó de cabeçalho contém um ponteiro para o último nó da lista.



   Essa implementação simplifica a representação de uma fila, pois ao invés de usar dois ponteiros (`q.inicio` e `q.fim`), precisamos apenas de um único ponteiro externo `lista`, para o nó de cabeçalho: `lista->next` aponta para o primeiro nó, e `lista->info` aponta para o último nó.

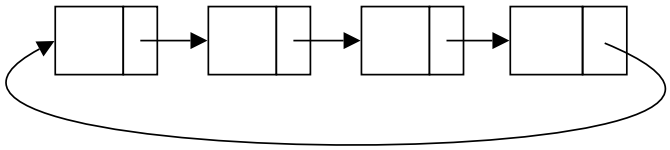
2. Outra possibilidade para o uso da parte `info` de um nó de cabeçalho é como um ponteiro para um nó "atual"na lista durante um processo de percurso. Isso eliminaria a necessidade de um ponteiro externo durante o percurso.

3.6 Outras estruturas de lista

Embora uma lista linear ligada seja uma estrutura de dados útil, ela apresenta várias deficiências. Nesta seção, vamos mostrar outros métodos de organizar uma lista e como eles podem ser usados para superar estas deficiências.

3.6.1 Listas circulares

Dado um ponteiro para um nó `p` numa lista linear, não podemos atingir nenhum dos nós que antecedem este nó. O que poderíamos fazer para resolver este problema? Uma solução que mostrou ser bastante útil é fazer uma pequena modificação na estrutura da lista: o campo `next` do último nó, ao invés de apontar para `NULL`, aponta para o primeiro nó da lista. Este tipo de lista é chamado de **lista circular**, e é mostrado abaixo:



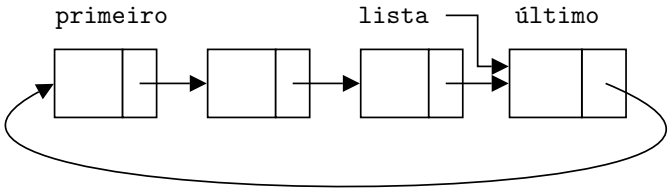
Com esta pequena modificação é possível agora atingir qualquer ponto da lista a partir de um outro ponto qualquer. Se começarmos em determinado nó e atravessarmos a lista inteira, terminaremos no ponto inicial.

Nós inicial e final de uma lista circular

Note que uma lista circular não tem um primeiro e um último nó. Precisamos portanto estabelecer um primeiro e último nós por convenção.

Já que é uma convenção, vamos estabelecer uma que seja inteligente: o ponteiro externo aponta para o último nó da lista. Porque? Isto permite o acesso ao último nó da lista (nó apontado por `p`) e ao primeiro nó da lista (nó apontado por `p->next`).bEssa convenção tem a vantagem de permitir a inclusão ou remoção de um nó a partir do início e do final de uma lista de forma fácil.

Além disso, estabelecemos que um ponteiro nulo representa uma lista circular vazia. Veja na figura abaixo:



A pilha como uma lista circular

Uma lista circular pode ser usada para representar uma pilha ou uma fila. Seja `pilha` um ponteiro para o último nó de uma lista circular, e adotemos a convenção de que o primeiro nó é o topo da pilha. Uma pilha vazia é representada por um ponteiro nulo. Com isso, vamos agora ver como ficam as operações `push` e `pop`. Chamei-as de `pushc()` e `popc()` para indicar que são funções para listas circulares.

```
void pushc (noptr &lista, int x)
{
  noptr p;

  p = new no;
  p->info = x;
  if (lista == NULL) // é o primeiro nó
    lista = p;
  else
    p->next = lista->next; // já existem outros nós
  lista->next = p;
}
```

```
int popc (noptr &lista, int &x)
{
  noptr p;

  if (lista == NULL) // a pilha já está vazia
    return -1;
  else // removendo nó ininicial
  {
    p = lista->next;
    x = p->info;
    if (p == lista) // somente um nó na pilha
      lista = NULL;
    else            // já existem outros nós na pilha
      lista->next = p->next;
    delete p;
    return 0;
  }
}
```

Da mesma forma que as estruturas anteriores, é importante que você desenhe uma lista circular e vá seguindo estes algoritmos passo a passo, para compreendê-los bem. Vou deixar como exercício a confecção de uma função main que leia vários dados os armazene na pilha usando a função `pushc()`. Depois, os dados devem ser desempilhados usando a função `popc()`. Apresente estes resultados na tela para que possam ser conferidos.

A fila como uma lista circular

É mais fácil representar uma fila como uma lista circular do que como uma lista linear, pois como uma lista linear precisamos de dois ponteiros, um para o início e outro para o fim. Entretanto, usando uma lista circular, uma fila pode ser especificada por um único ponteiro externo `fila` para esta lista. `fila` aponta para o último nó, e `fila->next` aponta para o início da fila.

Em uma fila, as remoções são realizadas no início da lista, de modo que podemos usar a função `popc()` definida acima para esta tarefa. Já as inserções são realizadas no final, e assim devemos implementar uma função que faça isso. Abaixo, temos o código da função `inserec()`, que realiza esta tarefa:

```
void inserec(noptr &lista, int x)
{
  noptr p;

  p = new no;
  p->info = x;
  if (lista == NULL) // primeiro nó a ser inserido
    lista = p;
  else               // demais nós
    p->next = lista->next;
  lista->next = p;
  lista = p;
}
```

Observe que `inserec()` é equivalente ao código:

```
pushc(lista,x);
lista = lista->next;
```

Ou seja, para inserir um elemento no final de uma fila circular, o elemento é inserido no início da fila e o ponteiro da lista circular avança um elemento para que o novo elemento ocupe o final (faça um diagrama para entender bem o que está acontecendo).

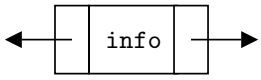
3.7 Listas duplamente ligadas

Embora uma lista circularmente ligada tenha vantagens sobre uma lista linear, ela ainda apresenta algumas deficiências:

- não se pode atravessar uma lista deste tipo no sentido contrário;
- não é possível eliminar um nó apontado por um ponteiro.

Nos casos em que tais recursos são necessários, a estrutura de dados adequada é uma **lista duplamente ligada**.

Cada nó numa lista deste tipo contém dois ponteiros: um para o nó à esquerda, e outro para o nó à direita, conforme mostra a figura abaixo:



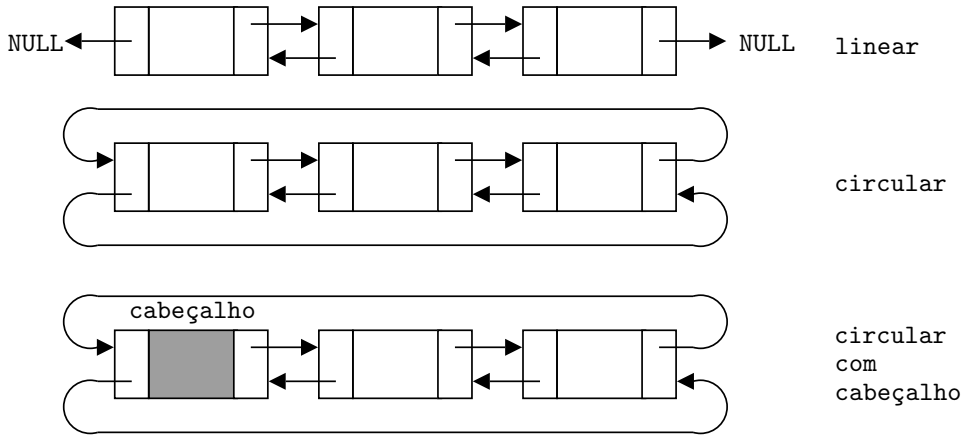
Em C++ um nó deste tipo seria declarado da seguinte maneira:

```
struct nod
{
  int info;
  nod *esq, *dir;
};

typedef nod* nodptr;
```

Uma propriedade conveniente destas listas é que, se `p` for um ponteiro para um nó qualquer, então:

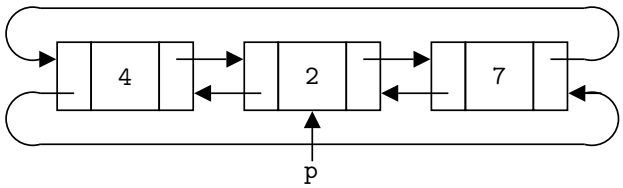
```
(p->dir)->esq = p = (p->esq)->dir
```

As listas duplamente ligadas podem ser lineares ou circulares e podem conter ou não um nó de cabeçalho. Veja abaixo alguns exemplos:

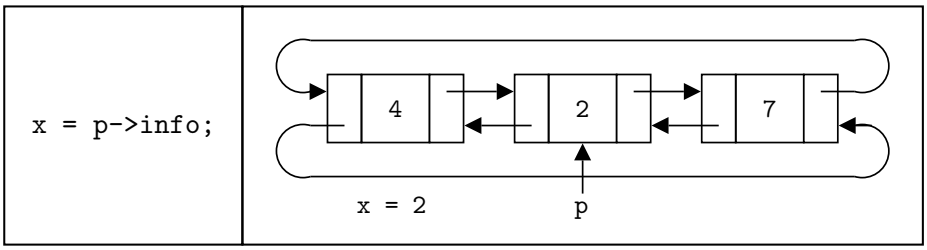


Removendo um nó apontado por `p`

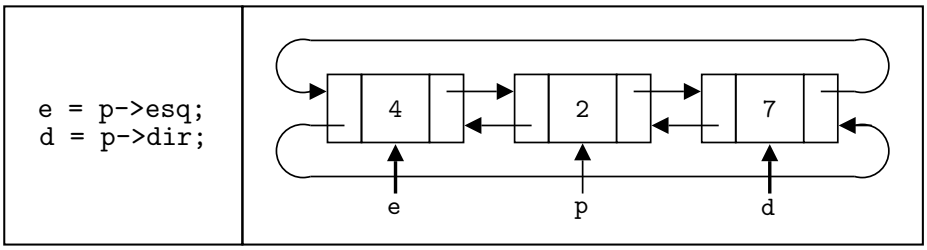
Agora vamos estudar algumas operações que podem ser realizadas sobre estas estruturas. A primeira delas é remover um nó apontado por `p` e armazenar o seu conteúdo em uma variável `x`. Como estamos em um tópico novo, vou mostrar de novo o procedimento passo a passo. Vamos supor inicialmente que temos uma lista circular duplamente ligada com alguns nós, e que desejamos remover o nó apontado por `p`, como mostrado na figura abaixo:



1. A primeira coisa a fazer é recuperar a informação contida no nó a ser removido:

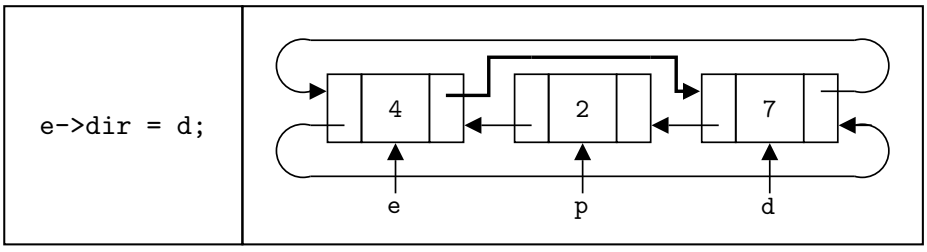


2. Depois, precisamos identificar quem são os nós à esquerda e à direita de `p`, para podermos refazer as ligações depois de o eliminarmos. Vamos chamar o nó à esquerda de `e`, e o nó à direita de `d`.

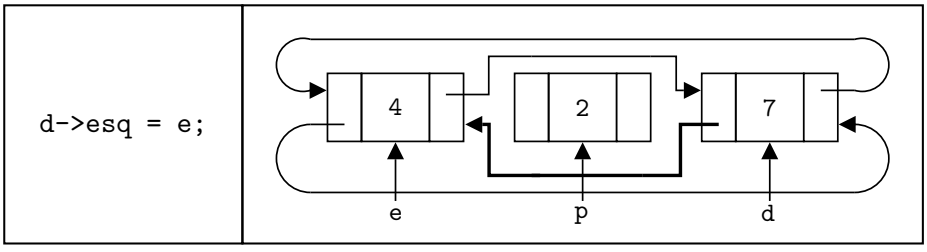


3. Agora precisamos ligar o nó `e` ao nó `d`. Vamos fazer isso em duas etapas:

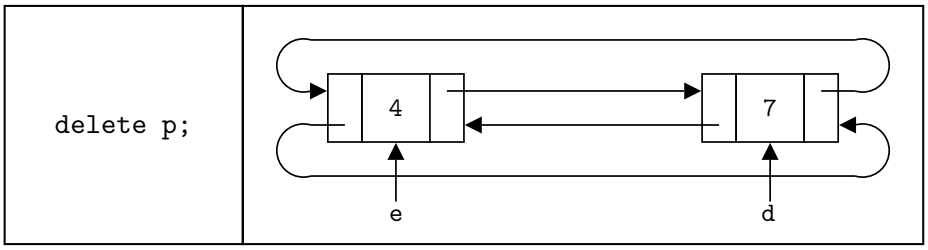
   (a) Primeiro, ligamos o campo dir do nó e ao nó d.



   (b) Depois ligamos o campo esq do nó d ao nó e.



4. Finalmente, liberamos a memória que estava sendo usada pelo nó p.



Se encapsularmos estes passos em uma função remove, teríamos o seguinte código:

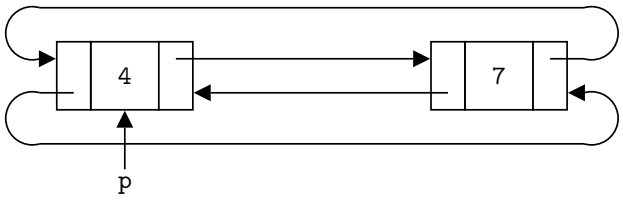
```
int removed(noptr &p, int &x)
{
  noptr e,d;

  if (p == NULL) // não há nada para remover
    return 1;
  else // removendo o nó apontado por p
  {
    x = p->info;
    e = p->esq;
    d = p->dir;

    if(d != NULL) // se existir um nó à direita de p
       d->esq = e;

    if(e!=NULL) // se existir um nó à esquerda de p
       e->dir = d;
            p=e;
    delete p;
    return 0;
  }
}
```

Ao invés da função retornar o conteúdo do nó, optei por fazer o retorno através de um parâmetro passado por referência. Então porque a função retorna um inteiro? Bem, a função retorna 0 se a remoção foi realizada com sucesso, e 1 se houve uma tentativa de remover um nó apontado por um ponteiro nulo, o que obviamente não é possível. Com isso, deixo o controle sobre a atitude a ser tomada (em caso de erro) para o programa ou função que a chamou.

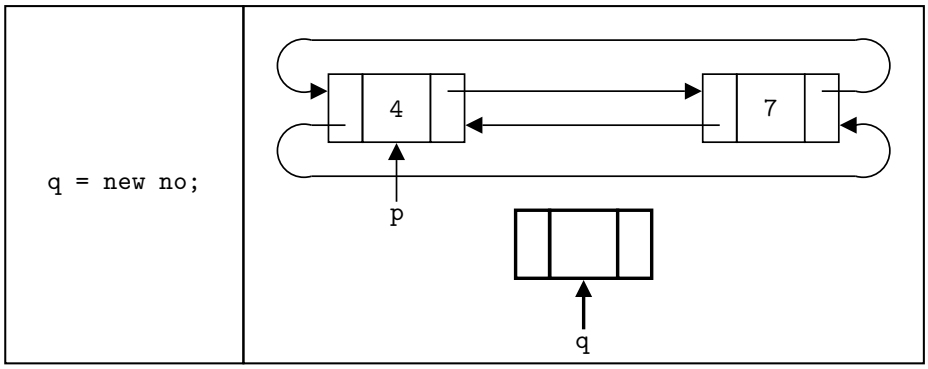
Importante: a forma como os desenhos foram feitos, pode gerar uma certa confusão. Tomando como exemplo o último diagrama, você pode achar que `e->dir` aponta para `d->esq`. Entretanto, é importante frisar que os ponteiros apontam para **nós** e não para **campos** dentro de nós. Desta forma, a interpretação correta da figura acima é que `e->dir` aponta para o **nó** `e`, e `d->esq` aponta para o **nó** `e`. Tome muito cuidado com isso!

Inserindo um nó à direita de p

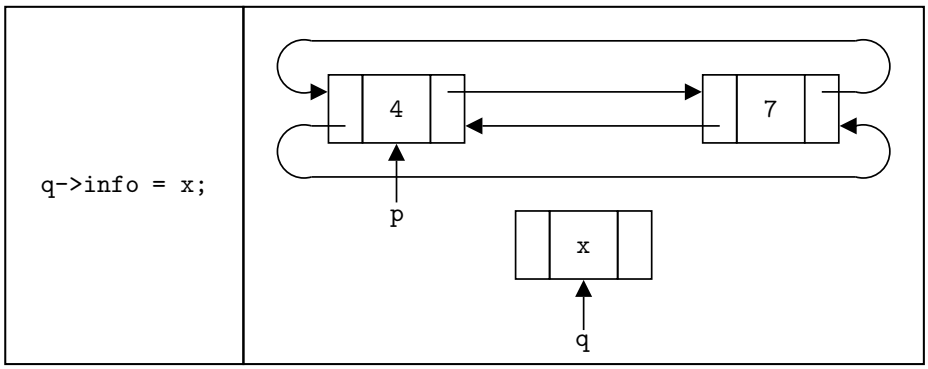
Outra operação possível é inserir nós à direita e à esquerda de um nó apontado por `p`. Vamos mostrar como inserir um nó à direita, e deixar como exercício o procedimento para inserir um nó à esquerda, ok? Como de costume, vamos supor inicialmente que temos uma lista com alguns nós, como mostrado abaixo:



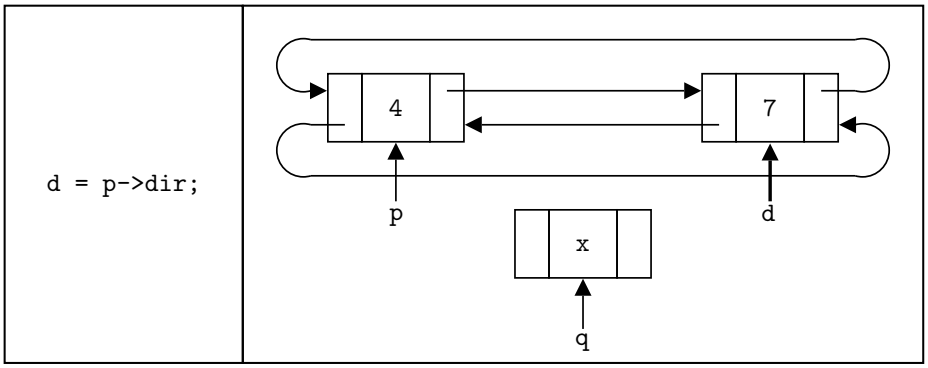
1. Primeiro, vamos alocar memória para um novo nó `q`.



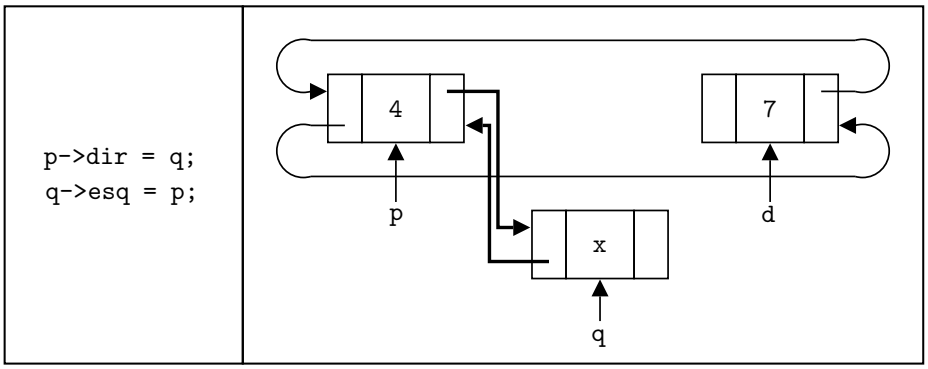
2. Depois, inserimos a informação a ser armazenada em seu campo `info`.



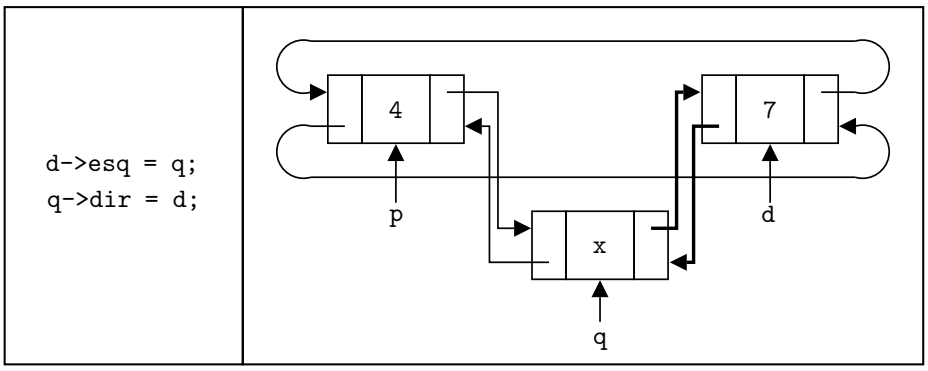
3. Identificamos o nó que está à direita de `p`. Vamos chamar este nó de `d`.



4. Efetuamos as ligações entre `p` e `q`.



5. Efetuamos as ligações entre `q` e `d`.



A função `inseredir()`, que encapsula este procedimento é mostrada abaixo:

```
int inseredir(noptr &p, int x)
{
  noptr q,d;

  if (p == NULL) // se p é NULL, não dá pra inserir à direita
    return 1;
  else // inserindo nó à direita de p
  {
    q = new no;
    q->info = x;
    d = p->dir;
    d->esq = q;
    q->dir = d;
    q->esq = p;
    p->dir = q;
    return 0;
  }
}
```

3.8 Exercícios

1. Implemente funções para executar as seguintes tarefas:

   (a) Retornar o número de elementos de uma lista.

   (b) Gerar uma lista resultante da concatenação de outras duas.

   (c) Eliminar o n-ésimo elemento de uma lista.

   (d) Retornar a soma dos elementos de uma lista.

   (e) Dados dois nós, inverter as posições dos mesmos. (Dica: não precisa inverter as posições, apenas os conteúdos dos campos info).

   (f) Deslocar um nó apontado por p, n posições à frente.

   (g) Inserir um nó depois do n-ésimo elemento de uma lista.

   (h) Verificar se existe um nó cujo campo info contém um determinado valor x.

   (i) Fazer uma cópia de uma lista.

   (j) Inverter uma lista.

   (k) Formar uma lista contendo a união dos elementos de duas listas.

   (l) Formar uma lista contendo a intersecção de duas listas.

   (m) Criar uma segunda cópia de uma lista.

   (n) Combinar duas listas ordenadas em uma única lista ordenada.

2. Faça um programa que crie uma lista ligada e insira alguns nós na mesma (você pode usar a função `push` para fazer isso). Depois o programa deve pedir ao usuário que digite o valor que os nós a serem eliminados deve conter, e chamar a função `removetudo` para executar a tarefa.

3. Escreva algoritmos para inserir e remover dados em uma fila de prioridade ascendente para os seguintes casos:

   (a) lista ordenada

   (b) lista desordenada

4. Reescreva as funçoes `empty`, `push`, `pop`, `insere` e `remove` considerando a presença de um nó de cabeçalho.

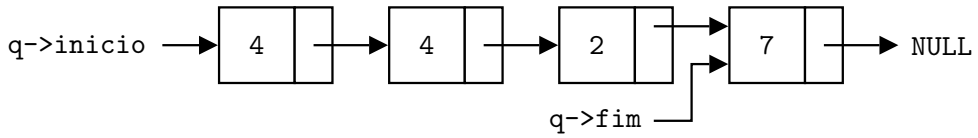
5. Reescreva as funções do exercício 1 pressupondo que cada lista contenha um nó de cabeçalho com o número de elementos da lista.

6. Escreva um algoritmo que retorne um ponteiro para um nó contendo o elemento `x` numa lista com um nó de cabeçalho. O campo info do cabeçalho deverá conter o ponteiro que atravessa a lista.

7. Faça um programa que leia vários dados e os armazene em uma pilha implementada como uma lista circular, usando a função `push`. Depois, os dados devem ser desempilhados usando a função `pop`. Apresente estes resultados na tela para que possam ser conferidos.

8. Faça uma função que insira um nó à esquerda de um nó apontado por `p` em uma lista duplamente ligada.

9. Dada a fila abaixo:



   Desenhe a nova configuração da mesma depois de executada a instrução

```
xyz(q,3);
```

   A função `xyz` é mostrada abaixo:

```
void xyz(fila &pq, int x)
{
  noptr p;

  p = new no;
  p->info = x;
  p->next = NULL;
  if (pq->fim == NULL)
    pq->inicio = p;
  else
    (pq->fim)->next = p;
  pq->fim = p;
}
```

A estrutura fila é mostrada abaixo:

```
struct fila
{
  noptr inicio;
  noptr fim;
};
```

10. O rail fence, cuja tradução literal é "paliçada", é uma cifra de transposição geométrica que foi muito utilizada na Guerra de Secessão norte-americana (1861-1865), quando tanto os confederados quanto os federalistas a utilizaram para cifrar mensagens.

    Seu funcionamento é bastante simples, e consiste em usar um padrão fixo em zig-zag. Digamos que a mensagem "`LULINHA PAZ E AMOR`" deva ser cifrada com uma Rail Fence de duas linhas:

```
L     L     N     A     A     E     M     R
   U     I     H     P     Z     A     O
```

    O texto cifrado resultante é obtido pela sequência de linhas, ou seja: `LLNAAEMR UIHPZAO`.

    Cifrando-se o mesmo texto com uma Rail Fence de 3 níveis obtemos LNAM UIHPZAO LAER. Veja abaixo:

```
L           N           A           M
   U     I     H     P     Z     A     O
      L           A           E           R
```

    Uma das etapas da codificação deste algoritmo consiste em gerar a sequência das linhas em que as letras serão armazenadas. Para uma rail fence de 3 linhas, devemos gerar a sequência:

    `1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 ...`

    e para uma rail fence de 4 linhas:

    `1 2 3 4 3 2 1 2 3 4 3 2 1 2 3 4 3 2 1 ...`

    Faça uma função que tome como entradas o número de linhas da rail fence (`N`), e o comprimento da string a ser codificada (`comp`) e mostre na tela a sequência de linhas correspondente, de tamanho `comp`. A função deve usar uma lista ligada circular para gerar a sequência de números.

    Por exemplo, se `N=3` e `comp=6`, a função deve imprimir:

    `1 2 3 2 1 2`

    O cabeçalho da função deve ser: `void gerasequencia(int N,int comp)`

    Você pode considerar que as funções `push(noptr &lista,int x)` e `insere(noptr &lista, int x)` estão disponíveis.

11. Seja o programa abaixo:

```
struct no
{
  int campo1;
  no *campo2;
};
typedef no* noptr;

void misterio(noptr e)
{
  while (e != NULL)
  {
    cout << e->campo1 << endl;
    e = e->campo2;
  }
}

int main()
{
  noptr p1, p2, pw;

  p1 = new no;
  p1->campo1 = 5;
  p1->campo2 = NULL;
  pw = new no;
  pw->campo1 = 7;
  pw->campo2 = p1->campo2;
  p1->campo2 = pw;
  p2 = new no;
  p2->campo2 = pw->campo2;
  p2->campo1 = 15;
  pw->campo2 = p2;

  misterio(p1);

  return 0;
}
```

   (a) Desenhe a lista ligada gerada por este programa.

   (b) O que será impresso na tela?

12. Implemente a função `place` para uma fila de prioridade ascendente, implementada a partir de uma lista duplamente ligada.

13. Faça uma função que tenha como entrada um ponteiro para uma lista ligada e um número inteiro `x`. A função deve retornar o número de elementos maiores que `x` na lista.

14. Seja o seguinte programa:

```
int main()
{
  noptr p,q;

  p=new no;
  p->info=4;
  p->next=NULL;
  q=new no;
  q->info=3;
  q->next=p;
  p=new no;
  p->info=2;
  p->next=q;
  q=new no;
  q->info=7;
  q->next=p;

  return 0;
}
```

    (a) Desenhe a lista gerada por este programa.

    (b) Qual ponteiro está apontando para o início da lista?

    (c) Escreva uma sequência de comandos para liberar a memória alocada.

15. Reimplemente a função `push` para uma lista circular com nó de cabeçalho, cujo campo `info` contém o número de nós atualmente na lista.

    Condições de projeto:

    - Assuma que a lista já está inicialmente criada com o nó de cabeçalho.
    - O ponteiro externo deve apontar para o nó de cabeçalho.

16. Faça uma função que insira um novo nó em uma lista ligada, de forma a mantê-la ordenada de forma descendente. Você pode se basear na função `place`, mostrada anteriormente. Considere que as funções `push` e `insertafter` estejam disponíveis.

17. A professora Lú deseja armazenar as notas dos alunos de forma a calcular rapidamente a média da turma. Para isto, pensou em guardá-las em uma lista ligada cujo nó de cabeçalho já mostra a média das notas. Implemente uma função que, ao inserir uma nova nota na lista, atualize automaticamente o nó de cabeçalho.

18. Faça uma função que tome um ponteiro para uma lista ligada e retorne a média dos elementos desta lista. A sua função deve ter o seguinte cabeçalho:

    `float media(noptr lista)`

19. Faça uma função que retorne a média dos elementos de uma lista ligada circular.

20. Faça uma função que remova todos os nós de uma lista ligada cujo campo `info` seja um número par. Assuma que as funções `pop()` e `delafter()` estejam disponíveis.

21. Reimplemente as funções `push` e `pop` para um no com a seguinte estrutura:

```
struct no
{
  char nome[200];
  int matricula;
  int idade;
};
```

22. Faça um programa que use as funções que você implementou para ler o nome e a idade de um conjunto de pessoas e gerar uma lista ligada.

    Depois o programa deve mostrar na tela, os dados inseridos, na seguinte ordem:

    (a) mulheres com menos de 18 anos
    (b) homens com menos de dezoito anos
    (c) mulheres com 18 anos ou mais
    (d) homens com 18 anos ou mais

23. Faça uma função que tome o ponteiro para uma lista ligada, gerada pelo exercício anterior, e motre o nome das pessoas com menos de 18 anos de idade. A função deve ter o seguinte cabeçalho:

```
void mostraMaiores(noptr lista)
```

24. Um sistema de tarifação telefônica mede o tempo gasto em ligações locais, interurbanas e internacionais, e calcula a conta total a ser paga pelo usuário. Supondo que o custo de uma ligação local seja de R$ 0,05/min, faça um programa que leia os seguintes dados de vários usuários:

    (a) nome
    (b) número do telefone
    (c) número de minutos em ligações locais

    e os armazene em uma lista ligada, juntamente com o valor total da conta (você deve apresentar a struct no nó).

    Depois o programa deve soltar uma listagem com o nome do usuário, número do telefone e total da conta, da seguinte maneira:

```
Ana 34719001 R$100,00
Beto 34719222 R$70,00
...
```

    Baseie seu programa nas funções `push` e `pop`.

25. Faça uma função que tome um ponteiro para uma lista ligada e inverta as posições do primeiro e último nós da mesma. (Dica: não precisa inverter as posições, apenas os conteúdos dos campos info).

    `void inverte(noptr &lista)`

26. Faça uma função que tome um ponteiro para uma lista ligada e duplique todos os nós cujo campo info seja par. Você pode considerar que as funções `push(lista,x)` e `insertafter(p,x)` já estejam implementadas.

    `void duplicapar(noptr &lista)`

27. Sejam `lista1` e `lista2`, duas listas ligadas. Estas duas listas possuem um nó de cabeçalho cujo campo `info` contém o número de elementos em cada lista. Faça uma função que gere uma terceira lista `lista3`, resultante da concatenação de `lista1` com `lista2`. Não se esqueça de atualizar o campo `info` do nó de cabeçalho de `lista3`.

28. Faça uma função que insira um elemento em uma lista duplamente ligada de forma a mantê-la sempre ordenada.

29. Suponha que `lista` seja uma fila de prioridade ascendente, implementada de forma circular. Escreva uma sequência de comandos para eliminar um elemento da lista.

Capítulo 4

Árvores Binárias

As listas ligadas apresentam grande flexibilidade sobre as representações contíguas de estruturas de dados, porém sua forte característica sequencial representa o seu ponto fraco. Esse arranjo estrutural das listas ligadas faz com que a movimentação ao longo delas seja feita com um nó por vez. Agora, veremos como é possível contornarmos este problema por meio do uso de estruturas de dados, como árvores, implementadas pelo uso de ponteiros e listas ligadas.

As estruturas de dados organizadas como árvores têm provado ser de grande valor para uma ampla faixa de aplicações, entre outras, representação de expressões algébricas, como um método eficiente para pesquisas em grandes conjuntos de dados, aplicações de inteligência artificial e algoritmos de compactação.

4.1 Definição

Uma **árvore binária** é um conjunto finito de elementos que está vazio ou particionado em três subconjuntos disjuntos:

- o primeiro contém um único elemento, denominado **raiz** da árvore;
- os outros dois subconjuntos são em si mesmo árvores binárias, chamadas de **subárvore esquerda** e **subárvore direita** da árvore original.

Observações:

- Uma subárvore esquerda ou direita pode estar vazia.
- Cada elemento de uma árvore binária é chamado **nó** da árvore.

Exemplo 4.1. Seja a árvore binária da Figura abaixo:

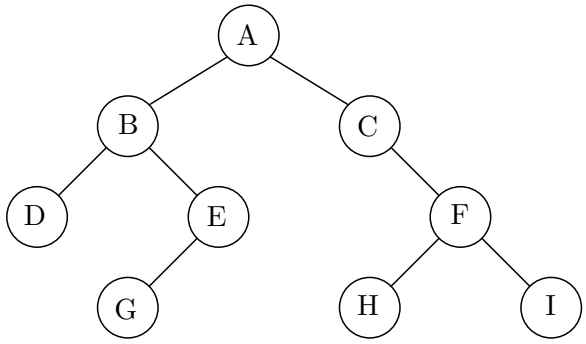


Figura 4.1: Exemplo de uma árvore binária

Sobre esta árvore podemos fazer as seguintes afirmações:

- esta árvore consiste de 9 nós, com raiz em A.
- sua subárvore esquerda está enraizada em B e sua subárvore direita, em C.
- a subárvore esquerda de C está vazia.
- idem para a subárvore direita de E.
- as árvores binárias enraizadas em D, G, H e I estão vazias.

4.2 Algumas definições importantes

a) **Pai e filho:** se A é a raiz de uma árvore binária e B é a raiz de sua subárvore direita, então A é *pai* de B, e B é *filho direito* de A. Por outro lado, se A é a raiz de uma árvore binária e C é a raiz de sua subárvore esquerda, então A é *pai* de C, e C é *filho esquerdo* de A.

b) **Folha:** é m nó sem filhos. Na Figura 4.1, são folhas os nós D, G, H e I.

c) **Ancestral e descendente:** o nó n_1 é um *ancestral* do nó n_2 (e n_2 é *descendente* de n_1), se n_1 for pai de n_2 ou pai de algum ancestral de n_2. Por exemplo, na árvore da Figura 4.1, A é um ancestral de G, e H é um descendente de C.

d) **Descendente esquerdo e direito:** um nó n_2 é *descendente esquerdo* do nó n_1 se n_2 for o filho esquerdo de n_1 ou um descendente do filho esquerdo de n_1. Um *descendente direito* pode ser definido da mesma forma. Na Figura 4.1, D e E são descendentes esquerdos de A, mas E é descendente direito de B.

e) **Irmãos:** dois nós são *irmãos* se forem filhos do mesmo pai. Na Figura 4.1, H e I são irmãos.

f) **Árvore estritamente binária:** é uma árvore binária na qual todo nó que não é folha tem subárvores esquerda e direita não vazias. Na Figura 4.2 tem-se um exemplo de uma árvore estritamente binária.

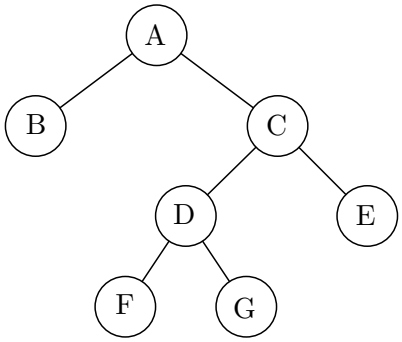


Figura 4.2: Exemplo de uma árvore estritamente binária.

f) **Nível:** o *nível* de um nó é definido como segue: a raiz da árvore tem nível 0, e o nível de qualquer outro nó na árvore é um nível a mais que o nível de seu pai. Na árvore da Figura 4.2, o nó A tem nível 0, os nós B e C têm nível 1, os nós D e E têm nível 2, e os nós F e G têm nível 3.

g) **Profundidade:** a profundidade de uma árvore binária é igual ao nível máximo de qualquer folha na árvore. Isso equivale ao tamanho do percurso mais distante da raiz arté qualquer folha. A árvore da Figura 4.2 tem profundidade 3.

h) **Árvore binária completa:** uma *árvore binária completa de profundidade d* é a árvore estritamente binária em que todas as folhas estão no nível d. A Figura 4.3 mostra um exemplo de uma árvore binária completa de nível 3.

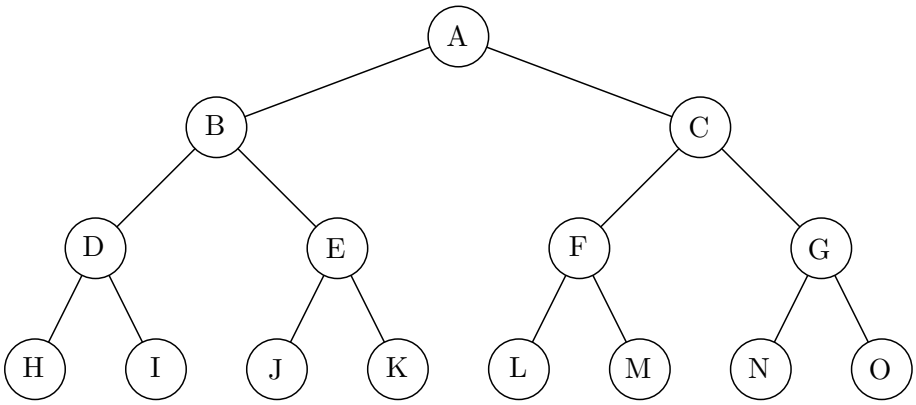


Figura 4.3: Exemplo de uma árvore binária completa de nível 3.

i) **Árvore binária quase completa:** uma árvore binária de profundidade d é uma árvore binária quase completa se:

1. todos os níveis, exceto prossivelmente o último, estão totalmente preenchidos.
2. todos os nós do último nível estão o mais à esquerda possível.

A árvore da figura a) não é quase completa, pois tem folhas nos níveis 1, 2 e 3, violando assim a primeira condição; a árvore da figura b) também não é quase

completa, pois os nós J e K violam a condição 2. As árvores das Figuras c) e d) são quase completas. Verifique cuidadosamente o porque, para entender bem o conceito.

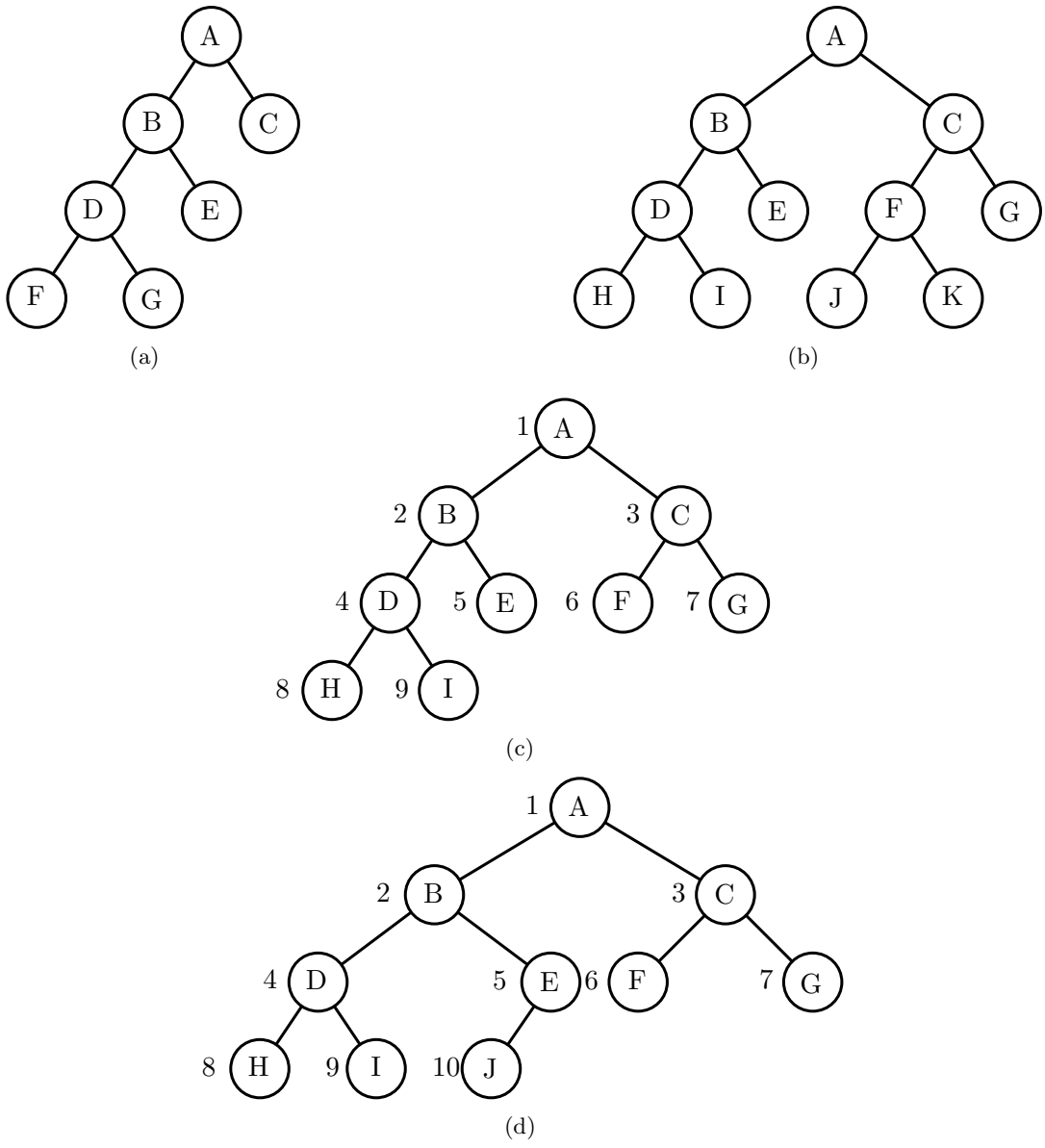


Figura 4.4: Numeração de nós para árvores binárias quase completas.

Numeração dos nós de uma árvore binária quase completa:

Os nós de uma árvore binária quase completa podem ser numerados da seguinte maneira:

- a raiz recebe o número 1;
- o filho esquerdo recebe um número que é o dobro do número do pai;

- o filho direito recebe um número que é o dobro do número do pai mais um.

As árvores c) e d) da Figura 4.4 ilustram esta técnica de numeração. Com isto, cada nó numa árvore binária quase completa recebe a atribuição de um número exclusivo, que define a posição do nó dentro da árvore.

4.3 Árvores de busca binária

A árvore de busca binária é uma estrutura que possui as seguintes propriedades:

- todos os itens da subárvore esquerda são menores do que a raiz;
- todos os itens da subárvore direita são maiores ou iguais à raiz;
- cada subárvore é também uma árvore de busca binária.

A Figura 4.5 mostra isto de forma esquemática:

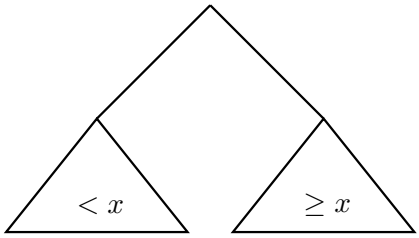
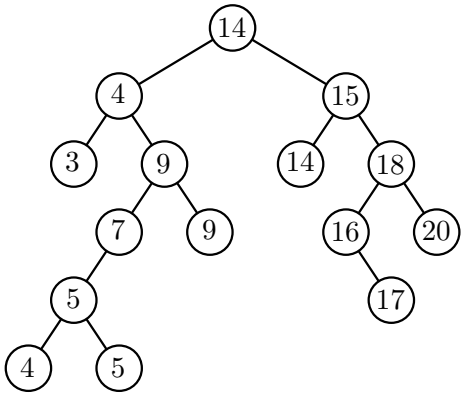


Figura 4.5: Esquema de uma árvore de busca binária.

Exemplo 4.2. Seja a seguinte sequência de valores de entrada:

$14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5$

A árvore binária para esta sequência é dada abaixo. Verifique com cuidado:



4.4 Manipulação de árvores de busca binária

Vamos apresentar agora algumas operações que podem ser realizadas sobre as árvores de busca binária. Estas operações permitem criar, inserir elementos e realizar consultas dos elementos de uma árvore de busca binária. A maioria dos algoritmos é implementada de forma recursiva, de modo que você deve ter este conceito bem claro. Se estiver em dúvida, estude bem este assunto antes de continuar.

Bom, chega de papo. Vamos começar com a representação de um nó da árvore, que é a base de tudo. Mãos à obra!

4.4.1 Declaração de um nó em uma árvore binária

Um nó deve ter um campo para a informação a ser armazenada,e ponteiros para seus filhos esquerdo e direito. Para diferenciar um nó de uma árvore de um nó de uma lista ligada, vamos adotar nomes diferentes para eles. Abaixo, tem-se a declaração de um nó de uma árvore:

```
struct treenode
{
  int info;
  treenode *esq;
  treenode *dir;
};

typedef treenode *treenodeptr;

int main()
{
  treenodeptr arvore = NULL;

  ...

  return 0;
}
```

Observe que uma árvore binária vazia é representada por uma variável ponteiro com conteúdo nulo: `treenodeptr arvore = NULL;`

4.4.2 Inserindo um elemento em uma árvore de busca binária

Para inserir um novo nó em uma árvore de busca binária, seguimos os seguintes passos:

- Verifica-se inicialmente se a árvore está vazia;
- Se estiver vazia, a inserção é feita na própria raiz da árvore;
- Se não estiver vazia, deve-se comparar o novo elemento com o elemento da raiz:
  - se for menor, a inserção será na subárvore esquerda;
  - se for maior ou igual, a inserção será na subárvore direita.

A função abaixo implementa esta idéia de forma bastante elegante, usando a recursão:

```
void tInsere(treenodeptr &p, int x)
{
  if (p == NULL) // insere na raiz
  {
    p = new treenode;
    p->info = x;
    p->esq = NULL;
    p->dir = NULL;
  }
  else
    if (x < p->info) // insere na subarvore esquerda
      tInsere(p->esq,x);
    else             // insere na subarvore direita
      tInsere(p->dir,x);
}
```

Exemplo 4.3. Para exemplificar o uso desta função considere o programa abaixo, que lê vários números e os insere na árvore. A entrada de dados termina quando for digitado o valor -1:

```
int main()
{
  int x=0; // variavel auxiliar para leitura de dados
  treenodeptr arvore = NULL; // armazena os numeros

  while (x != -1)
  {
    cout << "Digite elemento a ser inserido:\n";
    cin >> x;

    if (x != -1)
      tInsere(arvore,x);
  }

  return 0;
}
```

4.4.3 Pesquisa de um elemento

Considerando uma árvore de busca binária `T` e um elemento `x` a ser procurado entre seus nós, existem quatro possibilidades:

1. se `T` é uma árvore nula, então não há o que pesquisar;
2. se a raiz de `T` armazena o elemento `x`, a solução é imediata;

3. se `x` é menor que o valor armazenado na raiz de `T`, a busca deve prosseguir na subárvore esquerda de `T`;

4. se `x` é maior que o valor armazenado na raiz de `T`, a busca deve prosseguir na subárvore direitade `T`.

Com base nestas considerações é apresentada a função `tPesq` abaixo. Esta função retorna `NULL` se `x` não for encontrado na árvore, e retorna um ponteiro para a subárvore cuja raiz armazena o elemento procurado, se este for encontrado. A exemplo da função `tInsere`, esta é também uma função recursiva.

```
treenodeptr tPesq(treenodeptr p, int x)
{
  if (p == NULL) // elemento não encontrado
    return NULL;
  else
    if (x == p->info) // elemento encontrado na raiz
      return p;
    else
      if (x < p->info) // procura na subárvore esquerda
        return tPesq(p->esq,x);
      else             // procura na subárvore direita
        return tPesq(p->dir,x);
}
```

Exemplo 4.4. Vamos incrementar o programa do exemplo anterior: agora, inserimos um novo nó apenas se a informação não estiver já na árvore. Antes da inserção, o programa chama a função `tPesq` para verificar se o dado já consta da árvore, e só insere se `tPesq` retornar `NULL`:

```
int main()
{
  int x=0; // variavel auxiliar para leitura de dados
  treenodeptr arvore = NULL; // armazena os numeros

  while (x != -1)
  {
    cout << "Digite elemento a ser inserido:\n";
    cin >> x;

    if (x != -1)
      if (tPesq(arvore,x) == NULL)
        tInsere(arvore,x);
      else
        cout << "Elemento " << x << " já consta da árvore\n";
  }
  return 0;
}
```

4.4.4 Remoção de um elemento

Prepare-se, pois a remoção de um elemento de uma árvore é a tarefa mais trabalhosa e difícil de entender. Mas, como nem tudo neste mundo é fácil, vamos lá. O importante é manter a calma e tentar entender as coisas devagar, passo a passo.

Para facilitar as coisas, vamos supor inicialmente que o nó a ser removido é a raiz `T` da árvore. Neste caso, temos os seguintes cenários a serem avaliados:

- a raiz não possui filhos: neste caso, a solução é imediata; podemos removê-la e fazer `T` apontar para `NULL`;
- a raiz possui um único filho: podemos remover a raiz, e o nó filho toma o seu lugar;
- a raiz possui dois filhos: aí o caso complica. Não é possível que os dois filhos assumam o lugar do pai.

Para resolver o problema do nó com dois filhos, vamos pensar o seguinte: da definição de árvore binária, não é permitido ter na subárvore esquerda de qualquer nó nenhum elemento maior ou igual a ele. Da mesma forma, na subárvore direita não pode haver nenhum nó menor que a raiz. Assim, se tomarmos como herdeiro o menor elemento da subárvore direita deste nó, a árvore continua ordenada. Você pode verificar isso com a árvore do Exemplo 4.2.

O problema agora é encontrar o menor elemento da subárvore direita. Isto é mais fácil do que se imagina: basta observar que em uma árvore de busca binária, o menor elemento sempre se encontra no nó mais à esquerda. Então, antes de começar a eliminar os nós, vamos propor uma função que recebe um ponteiro `raiz` para uma árvore não vazia, e em seguida faz uma pesquisa para identificar o nó que armazena seu menor elemento. Se o nó for achado, a `raiz` é substituída por esse valor, e a função retorna um ponteiro para o nó encontrado. Observe:

```
treenodeptr tMenor(treenodeptr &raiz)
{
  treenodeptr t;

  t = raiz;
  if (t->esq == NULL) // encontrou o menor valor
  {
    raiz = raiz->dir;
    return t;
  }
  else // continua a busca na subárvore esquerda
    return tMenor(raiz->esq);
}
```

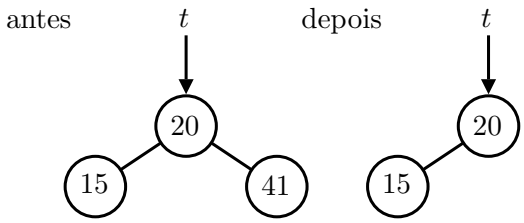
Agora, usando a função `tMenor`, ficou fácil remover um nó com dois filhos:

```
p = tMenor(raiz->dir); // desliga o nó com o menor elemento
raiz->info = p->info;   // armazena o valor na raiz da árvore
delete p;               // libera o nó removido
```

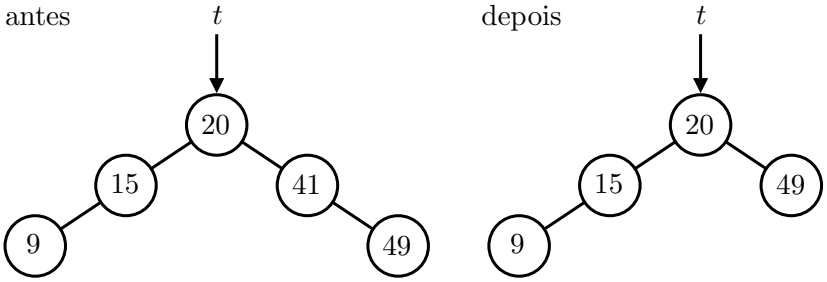
Juntando tudo numa função só, podemos criar uma função que remove um nó qualquer de uma árvore de busca binária. A seguinte função remove um nó cujo campo `info` contém o valor especificado por x. Retorna 0 caso a remoção tenha sido realizada com sucesso, e 1 caso contrário. Veja:

```
int tRemove(treenodeptr &raiz, int x)
{
  treenodeptr p;

  if (raiz == NULL) // árvore vazia
    return 1;
  if (x == raiz->info)
  {
    p = raiz;
    if (raiz->esq == NULL) // a raiz não tem filho esquerdo
      raiz = raiz->dir;
    else
      if (raiz->dir == NULL) // a raiz não tem filho direito
        raiz = raiz->esq;
      else // a raiz tem ambos os filhos
      {
        p = tMenor(raiz->dir);
        raiz->info = p->info;
      }
    delete p;
    return 0;
  }
  else
    if (x < raiz->info)
      return tRemove(raiz->esq,x);
    else
      return tRemove(raiz->dir,x);
}
```

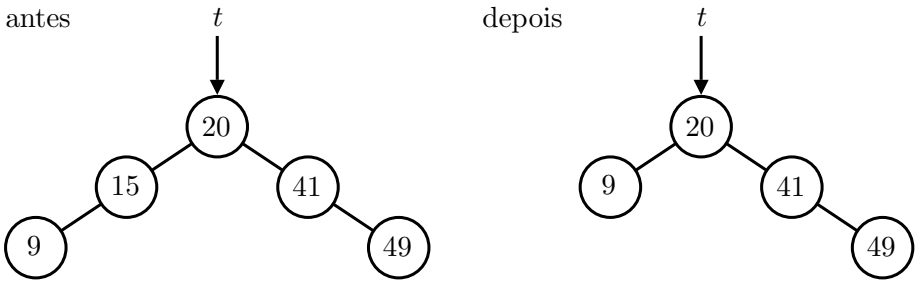
Sugiro que você estude com cuidado esta função, pois não é muito fácil de entender. Na verdade, algoritmos recursivos são bastante complicados, e por isso merecem atenção especial. Deixe a preguiça de lado, arregace as mangas e vamos lá. Pra ajudar um pouco, são listados na Figura 4.6 abaixo, os principais cenários que podem ocorrer durante o processo de remoção de nós em uma árvore de busca binária:



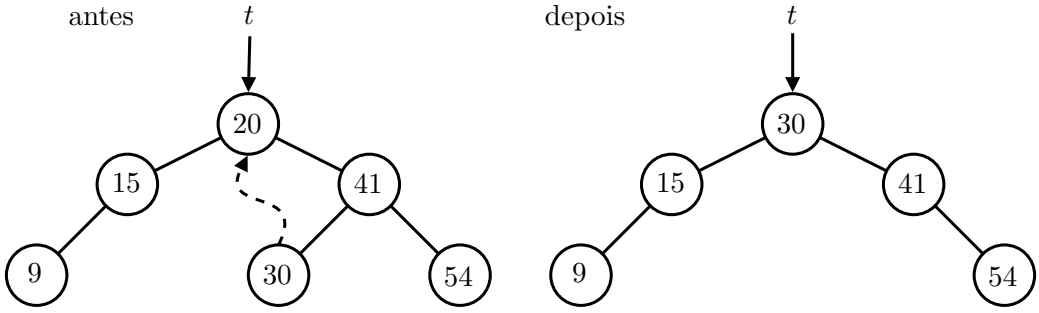
(a) Remoção do nó (41) sem nenhum filho.



(b) Remoção do nó (41) sem filho esquerdo.



(c) Remoção do nó (15) sem filho direito.



(d) Remoção do nó (20) com dois filhos.

Figura 4.6: Principais cenários na remoção de nós.

4.5 Percursos em árvores de busca binárias

Realizar um percurso em uma árvore binária consiste em visitar todos os seus nós de forma sistemática, fazendo um certo processamento para cada um deles. Ainda, cada nó deve ser processado uma única vez.

Existem duas formas de se fazer isto:

- **Percurso em profundidade:** todos os descendentes de um nó filho são proces-

sados antes do próximo nó filho. Pode ocorrer de três formas:

  - pré-ordem
  - em-ordem
  - pós-ordem

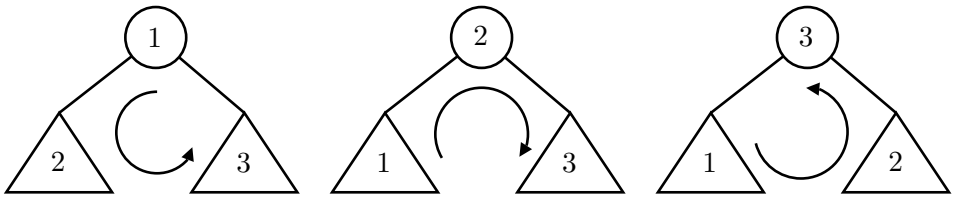
- **Percurso em largura:** o processamento ocorre de forma horizontal da raiz para todos os nós filhos, depois para os filhos destes nós, e assim por diante. Em outras palavras, cada nível da árvore é processado antes que o próximo nível seja iniciado.

4.5.1 Percursos em profundidade

Como vimos aí em cima, existem três tipos de percurso em profundidade, que são descritos de forma esquemática na Tabela 4.1 e na Figura 4.7.

Tabela 4.1: Tipos de percursos em profundidade.

Tipo	Sequência
pré-ordem	raiz (R) - folha esquerda (E) - folha direita (R)
em-ordem	folha esquerda (E) - raiz (R) - folha direita (D)
pós-ordem	folha esquerda (E) - folha direita (D) - raiz (R)



(a) Pré-ordem. (b) Em-ordem. (c) Pós-ordem.

Figura 4.7: Tipos de percursos em profundide.

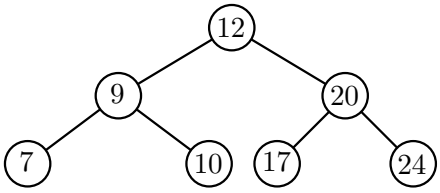
Bom, agora que já sabemos o basico, vamos ver como se implementam funções para realizar estes percursos. Como todas as coisas que fizemos neste tópico, estas são funções recursivas, extremamente simples e elegantes, mas difíceis de rastrear. Estude com afinco.

```
void preOrdem (treenodeptr arvore)
{
  if (arvore != NULL)
  {
    cout << arvore->info << endl;
    preOrdem(arvore->esq);
    preOrdem(arvore->dir);
  }
}
```

```
void emOrdem (treenodeptr arvore)
{
  if (arvore != NULL)
  {
    emOrdem(arvore->esq);
    cout << arvore->info << endl;
    emOrdem(arvore->dir);
  }
}
```

```
void posOrdem (treenodeptr arvore)
{
  if (arvore != NULL)
  {
    posOrdem(arvore->esq);
    posOrdem(arvore->dir);
    cout << arvore->info << endl;
  }
}
```

Exemplo 4.5. Para entender como funcionam estas funções, verifique os resultados dos três tipos de percurso sobre a árvore da Figura abaixo. Vá acompanhando os resultados com o código em mãos pra entender o que está acontecendo; pelo menos foi o jeito que eu mesmo entendi a história toda.



pré-ordem: 12, 9, 7, 10, 20, 17, 24
em-ordem: 7, 9, 10, 12, 17, 20, 24
pós-ordem: 7, 10, 9, 17, 24, 20, 12

4.5.2 Percurso em largura

No percurso em largura, também conhecido por percurso em nível, os nós são acessados nível por nível, ou seja, de cima para baixo, da esquerda para a direita.

Podemos usar uma fila para realizar o percurso em nível, adotando o seguinte procedimento:

- em um primeiro instante, a fila contém apenas o nó raiz;

- depois, enquanto a fila não se tornar vazia, retira-se um nó, cujos filhos deverão ser colocados na fila;
- em seguida, o nó retirado da fila é exibido.

```
void emNivel(treenodeptr t)
{
  treenodeptr n;
  noptr q = NULL;

  if (t != NULL)
  {
    insere(q,t);
    while (isempty(q) != true)
    {
      n = remove(q);
      if (n->esq != NULL)
        insere(q,n->esq);
      if (n->dir != NULL)
        insere(q,n->dir);
      cout << n->info << endl;
    }
  }
}
```

É bom fazer alguns comentários sobre esta função:

- Veja que temos um ponteiro para um nó de uma árvore (`treenodeptr n`) e um ponteiro para um nó de uma fila (`noptr q`).
- Cada nó da *fila* deve ter em seu campo `info`, um ponteiro para um nó de uma árvore. Assim, podemos declarar o nó da fila como:

```
struct no
{
  treenodeptr info;
  no *next;
};

typedef no* noptr;
```

4.5.3 Destruindo uma árvore

Você já deve ter esbarrado neste cenário: um programa funciona muito bem da primeira vez que roda. Depois que você termina o que está fazendo, fecha ele e vai fazer outra coisa. Quando volta pra continuar o trabalho, ele trava e você tem que reiniciar a máquina. Por que isso acontece? Uma das principais razões para isso é o que chamamos

de *vazamento de memória*, que acontece quando alocamos determinada quantidade de memória e esquecemos de desalocá-la. Assim, quando o programa termina, não devolve toda a memória utilizada para o sistema operacional. Dá pra perceber que a cada vez que rodamos o maldito programa, a memória disponível vai diminuindo, diminuindo, até acabar. Daí o computador trava.

Por esta razão, é de extrema importância elaborarmos programas que não deixem "lixo" na memória. Vamos então mostrar a função `tDestruir`, que desaloca todos os nós de uma árvore de busca binária, usando um percurso em pós-ordem.

```
void tDestruir (treenodeptr &arvore)
{
  if (arvore != NULL)
  {
    tDestruir(arvore->esq);
    tDestruir(arvore->dir);
    delete arvore;
  }
  arvore = NULL;
}
```

Observe que não dá pra implementar a função `tDestruir` usando os percursos em-ordem e pós-ordem, pois isso geraria nós órfãos. Verifique este ponto.

4.6 Compactação de dados com códigos de Huffman

Você já deve ter manipulado arquivos compactados nos formatos zip, tar.gz entre outros. Como é possível transformar um arquivo de alguns megabytes em outro de algumas centenas de kilobytes, e depois voltar exatamente ao arquivo original? Bem, nesta seção irei apresentar uma forma de fazer isso. Interessado? Então vamos lá.

A forma mais fácil de entender algumas coisas, a meu ver é com exemplos. Assim, vamos a ele: cosidere a existência de uma coleção de dados com cerca de 50000 ocorrências dos seguintes caracteres: a, e, i, o, u, !. Assuma ainda que os caracteres ocorrem com as seguintes frequências percentuais:

Caractere	a	e	i	o	u	!
Frequência (%)	52	8	12	11	7	10

Para armazenar/transmitir estes seis caracteres precisamos de 3 bits. Desta forma, para armazenar/transmitir 50000 caracteres, precisaremos de 150000 bits.

Uma forma alternativa (e mais inteligente também) é representar os caracteres que ocorrem com mais frequência com um número menor de bits do que os que ocorrem com menos frequência. Desta forma, na maior parte das vezes, estaremos armazenando/transmitindo poucos bits. Não é genial? Simples e inteligente. Vamos ver um exemplo:

Caractere	a	e	i	o	u	!
Sequência variável de bits	1	1110	010	100	0110	000

Se usarmos este esquema de codificação, para armazenar/transmitir estes mesmos 50000 caracteres iremos precisar de 105500 bits, uma economia de aproximadamente 30%, como mostra a tabela abaixo:

Caractere	a	e	i	o	u	!
Número de bits	1	4	3	3	4	3
Frequência (%)	52	8	12	11	7	10
Total de bits/caractere	26000	16000	18000	16500	14000	15000

A idéia é boa, mas existe um problema: o código deve ser inequivocamente decodificável, isto é, uma sequência de bits deve corresponder a uma única sequência de caracteres, ou seja, não pode haver dúvidas na hora da decodificação. Um cara chamado Huffman descobriu um jeito de fazer isso de forma fácil e elegante.

As entradas para o algoritmo são:

- n, o número de caracteres;
- $freq$, um vetor contendo a frequência de ocorrência de cada caractere.

A saída do algoritmo é:

- *code*, um vetor de strings com os bits atribuídos a cada caractere de entrada.

O algoritmo controla a construção de uma árvore com base em uma lista `rootnodes` de nós, que são as raízes de árvores binárias. Em um primeiro momento, `rootnodes` terá `n` raízes e cada uma delas será rotulada com a frequência dos caracteres. As raízes são classificadas em ordem crescente de acordo com a frequência e não têm filhos.

Em seguida, os nós com menor frequência são combinadas, gerando uma árvore que tem em seu nó raiz a soma das frequências dos dois nós, e como folhas, os nós originais. Os nós combinados são retirados da lista `rootnodes`, e a raiz da árvore recém criada é colocada em `rootnodes`, de modo a manter a lista ordenada.

Este processo continua até que `rootnodes` contenha apenas um nó.

Depois de construída a árvore, colocamos um valor de bit em cada ramo da árvore: começando pela raiz, colocamos 0 para o ramo esquerdo e 1 para o ramo direito, repetindo este padrão para todos os nós da árvore.

Finalmente, determinamos o código para cada caractere. Para isso, usamos o vetor `position` que guarda ponteiros para os nós que representam cada caracter. Usando este vetor, partimos das folhas (que são os nós cujos ponteiros estão em `position`) e vamos anotando os bits de cada ramo até chegarmos ao nó raiz (`rootnodes`).

Abaixo, temos o código do codificador:

```
// Codificador de Huffman
// Suponha que temos n caracteres na entrada, e que
// a frequência de cada um deles está armazenada no
// vetor freq.

// Criando lista ordenada de raízes de árvores binárias
for (i=0;i<n;i++)
{
  p = new treenode;
  p->info = freq[i];
  p->father = NULL;
  position[i] = p;
  insereOrdenado(rootnodes,p);
}

// Combinando os nós
while (rootnodes->next != NULL)
{
  p1 = pop(rootnodes);
  p2 = pop(rootnodes);

  p = new treenode;
  p->info = p1->info+p2->info;
  p->esq = p1;
  p->dir = p2;
  p1->father = p;
  p2->father = p;
  insereOrdenado(rootnodes,p);
}

// Árvore criada. Construindo os códigos
root = pop(rootnodes);
for (i=0;i<n;i++)
{
  comp[i] = 0;
  p = position[i];
  j = 0;
  while (p != root)
  {
    if (isesq(p))
      code[i][j++] = '0';
    else
      code[i][j++] = '1';
    comp[i]++;
    p = p->father;
  }
}
```

Observações

- Por causa do tamanho do código, omiti as declarações de variáveis e de funções, mas acho que dá pra entender a idéia.

- Os nós da árvore binária contém um ponteiro `father` apontando para o nó pai do nó atual, além dos ponteiros para os filhos esquerdo e direito.

- a função `isesq(p)` verifica se o nó `p` é ou não filho esquerdo de um outro nó. Isto fica fácil quando se tem um campo `father` apontando para o nó pai.

- Observe que, da forma como foi implementado, o programa fornece o código invertido, ou seja, de trás pra frente. De qualquer forma, inverter a saída é uma coisa trivial.

Exemplo de funcionamento do algoritmo de Huffman

Tá bom, eu sei que você tá achando tudo isso meio difícil. Nada como um exemplo pra clarear as idéias. Vamos ver como chegar ao código de Huffman para o exemplo que estamos usando até agora. Só pra relembrar, temos os seguintes caracteres, com as respectivas frequências:

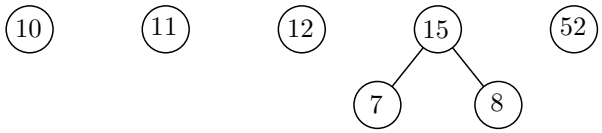
Caractere	a	e	i	o	u	!
Frequência (%)	52	8	12	11	7	10

Vamos seguir o programa passo a passo:

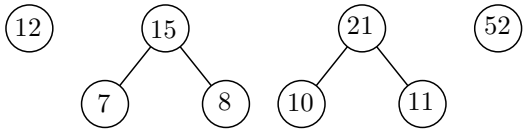
- `Criando lista ordenada de raízes de árvores binárias`

- `Combinando os nós`

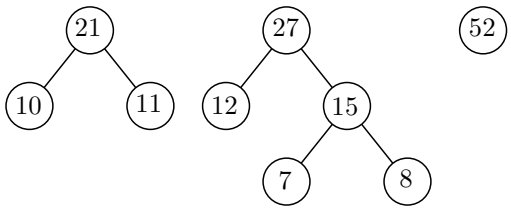
  - Primeira vez



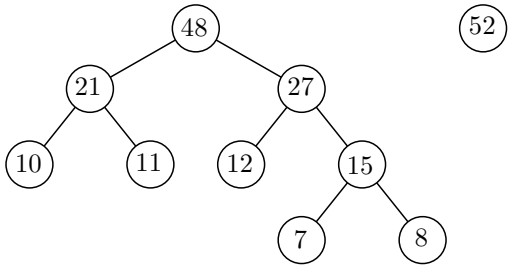
  - Segunda vez



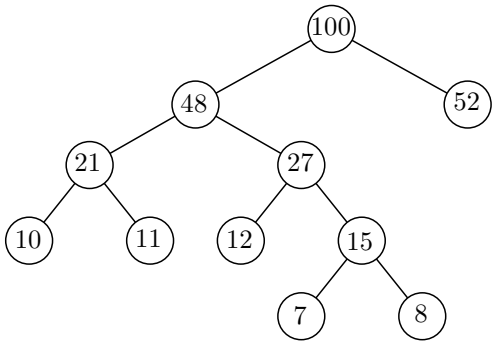
– Terceira vez



– Quarta vez

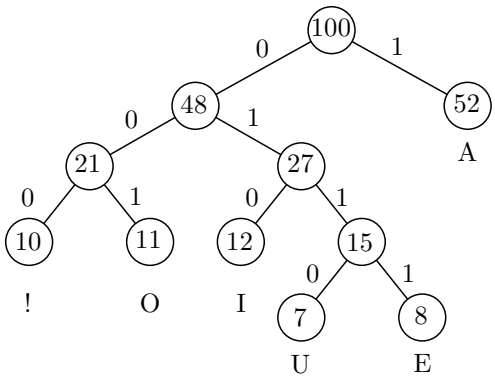


– Última vez



- `Árvore criada. Construindo os códigos`

  Primeiro atribuímos o bit 0 a todos os ramos esquerdos, e o bit 1 a todos os ramos direitos, como mostra a figura abaixo:



  Depois é só verificar os bits que estão no caminho de cada nó folha até a raiz. Fazendo isto, temos o resultado mostrado na tabela abaixo:

Caractere	a	e	i	o	u	!
Sequência de bits	1	1110	010	100	0110	000

Observe que esta é a mesma tabela mostrada anteriormente.

4.7 Exercícios

1. Uma árvore binária tem dez nós. Os passeios em-ordem e pré-ordem da árvore são mostrados a seguir. Desenhe a respectiva árvore.

   Pré-ordem: J C B A D E F I G H

   Em-ordem: A B C E D F J G I H

2. Uma ávore binária tem sete nós. Os passeios pré-ordem e pós-ordem da árvore são mostrados abaixo. É possível desenhar a árvore? Se não, justifique sua resposta.

   Pré-ordem: G F D A B E C

   Pós-ordem: A B D C E F G

3. Qual é o menor número de níveis que uma árvore de 42 nós pode ter?

4. Qual é o número máximo de nós no nível cinco de uma árvore binária?

5. Escreva um algoritmo que conte o número de nós em uma árvore binária.

6. Escreva um algoritmo que conte o número de folhas em uma árvore binária.

7. Faça uma função que calcule a média de todos os nós de uma árvore, usando um passeio em pré-ordem.

8. Escreva um algoritmo para excluir todas as folhas de uma árvore binária, deixando a raiz e os nós intermediários no respectivo lugar. *Dica: use o passeio pré-ordem.*

9. Escreva um algoritmo que verifique se uma árvore binária é completa ou não.

10. Reescreva o algoritmo de passeio pré-ordem usando uma pilha em vez da recursão.

11. Reescreva o algoritmo de passeio em-ordem usando uma pilha em vez da recursão.

12. Escreva um algoritmo que cria uma imagem espelho de uma árvore binária, isto é, todos os filhos à esquerda tornam-se filhos à direita, e vice-versa.

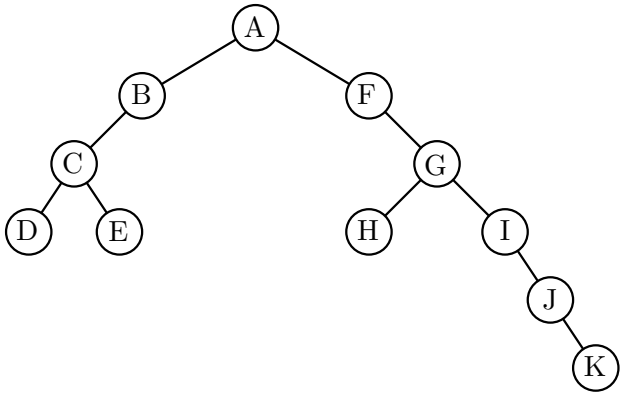
13. Ache a raiz de cada uma das seguintes árvores binárias:

árvore com percurso pós-ordem	F C B D G
árvore com percurso pré-ordem	I B C D F E N
árvore com percurso em-ordem	C B I D F G E

14. Mostre o resultado dos três percursos em profundidade e do percurso em largura para a árvore binária abaixo:

15. Qual a altura máxima e mínima de uma árvore de 28 nós?

16. Em uma árvore binária, qual é o número de máximo de nós que pode ser achado nos níveis 3, 4 e 12?



17. Qual é o menor número de níveis que uma árvore de 42 nós pode ter?

18. Dados os códigos:

Caractere	A	E	I	O	U
Codificação	00	01	10	110	111

Decodifique as sequências

- 1 1 0 1 1 0 1 1 1 0 1
- 1 0 0 0 1 1 0 1 1 1
- 0 1 0 1 0 1

19. Construa a respectiva árvore de Huffman para os caracteres e para as frequências de dados relacionadas a seguir. Encontre também os códigos de Huffman para estes caracteres.

Caractere	H	V	X	R	K	N
Frequência	29	22	7	19	4	19

20. Para um alfabeto de 4 símbolos (A, B, C, D) tem-se as seguintes informações:

- o símbolo B ocorre com frequência 2 vezes maior que o símbolo A;
- o símbolo C ocorre com frequência 3 vezes maior que o símbolo A;
- o símbolo D ocorre com frequência 4 vezes maior que o símbolo A;

Construa a árvore e os códigos de Huffman para este alfabeto.

Dica: lembre-se que a soma das frequências relativas dos 4 símbolos deve ser igual a 100.

21. Decodifique a mensagem abaixo, sabendo-se que ela foi cifrada a partir de um código de Huffman, que por sua vez foi gerado a partir dos caracteres abaixo:

Mensagem: 000100011010101001000111101111110

Caractere	A	E	F	I	L	N	T	Z
Frequência	31	18	1	6	11	13	17	3

22. Decodifique a mensagem abaixo, sabendo-se que ela foi cifrada a partir de um código de Huffman, que por sua vez foi gerado a partir dos caracteres abaixo:

    Mensagem: 00110101001111101110111111100111

Caractere	A	C	D	E	I	N	P	R
Frequência	23	11	4	10	15	3	21	13

23. Faça uma função que calcule a média de todas as folhas de uma árvore. Use uma modificação do percurso em pré-ordem.

24. Faça uma função que calcule a média de todos os nós que não são folhas de uma árvore binária. Use uma modificação do percurso em pós-ordem.

25. Faça uma função que conte quantos nós de uma árvore são folhas e quantos não são folhas.

26. Suponha que um programador decidiu armazenar as idades dos usuários de uma locadora de filmes em uma árvore binária. Como você deve saber, existem filmes que não devem ser assistidos por menores de 18 anos. Para saber se vale a pena investir na compra de tais filmes, foi pedido ao programador que adicionasse ao programa, uma funcionalidade que verificasse o número de pessoas com idade maior ou igual a 18 anos. O problema é que este programador foi demitido e o problema ficou para você.

    Faça então uma função que tome como entrada um ponteiro para uma árvore e retorne o número de nós cujo campo info tenha valor maior ou igual a 18. O cabeçalho da sua função deve ter a seguinte forma:

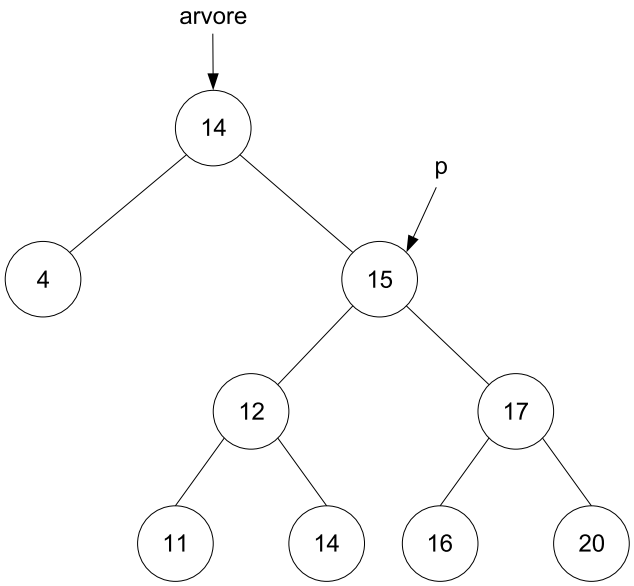
    ```
    int nMaiores(treenodeptr arvore)
    ```

    Observação: use uma variante do percurso em-ordem.

```
void emOrdem (treenodeptr arvore)
{
  if (arvore != NULL)
  {
    emOrdem(arvore->esq);
    cout << arvore->info << endl;
    emOrdem(arvore->dir);
  }
}
```

27. Faça uma função que zere todos os nós de uma árvore binária cujo campo `info` seja igual a um certo valor `x`, fornecido pelo usuário.

28. Seja a árvore binária abaixo:



Desenhe a mesma depois que o comando

```
tMenor(p);
```

for executado.

```
treenodeptr tMenor(treenodeptr &raiz)
{
  treenodeptr t;

  t = raiz;
  if (t->esq == NULL) // encontrou o menor valor
  {
    raiz = raiz->dir;
    return t;
  }
  else // continua a busca na subárvore esquerda
    return tMenor(raiz->esq);
}
```

Capítulo 5

Árvores AVL

O que acontece se inserirmos os elementos em uma árvore binária de forma ordenada (em ordem crescente ou decrescente)? A árvore ficaria como na figura abaixo:

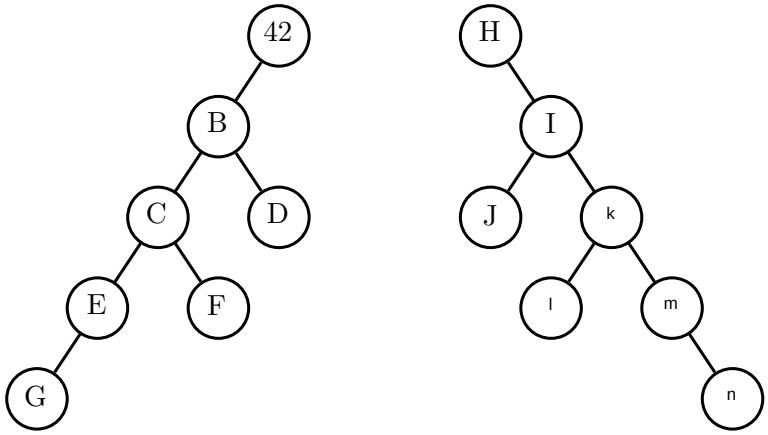


Figura 5.1: Exemplo de uma árvore binária

Referências Bibliográficas

[1] www.cplusplus.com

[2] SAVITCH, Walter. C++ absoluto. Addison Wesley, 2004.

[3] TENENBAUM, Aaron, LANGSAM, Yedidyah, AUGENSTEIN, Moshe J. Estruturas de dados usando C. Makron Books. 1995.

[4] MORAES, Celso Robert. Estruturas de dados e algoritmos - uma abordagem didática. Futura, 2003.