Árvores Binárias de Busca

0. Um breve comentário sobre os algoritmos de busca em tabelas

De uma maneira geral, realizam-se operações de busca, inserção e remoção de elementos numa tabela.

A busca sequencial tradicional é O(N). Não é eficiente, mas permite inserções e remoções rápidas. A inserção pode ser feita no final da tabela, pois a ordem não precisa ser preservada. A remoção pode ser feita simplesmente pela substituição do elemento removido por um valor especial que não faz parte da tabela. Entretanto, é importante notar que uma inserção ou remoção é quase sempre precedida por uma busca.

A busca binária é O(logN). É muito eficiente, mas a tabela deve estar em ordem crescente ou decrescente. Portanto inserções e remoções são muito ineficientes. Para inserir ou remover mantendo a ordem, é necessário deslocar parte da tabela.

A busca em tabela hash sequencial depende da função de hash e da variedade dos dados. Uma vantagem é que permite inserção de novos elementos. A remoção não é permitida, pois altera a estrutura da tabela.
No caso geral, pouco se pode afirmar sobre a eficiência do hash em tabela sequencial. No pior caso é O(N).
Outro inconveniente é que no hash a tabela ocupa mais espaço que a quantidade de elementos.

No caso do hash com lista ligada, inserção e remoção são facilitadas com a ocupação ideal de memória. Entretanto no pior caso continua sendo O(N).

A situação ideal seria um algoritmo que tivesse a eficiência da busca binária O(logN), permitisse inserções e remoções rápidas e que a tabela ocupasse somente o espaço necessário.

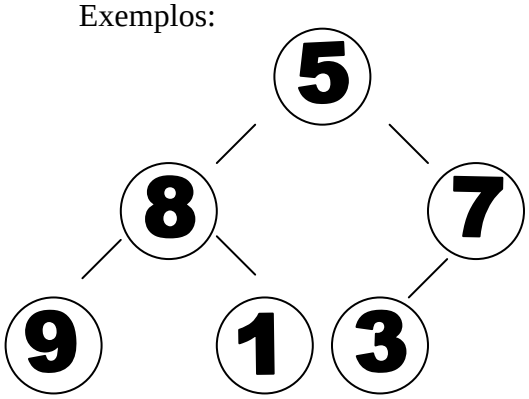
Isso é conseguido quando a tabela tem uma estrutura em **árvore de busca**.

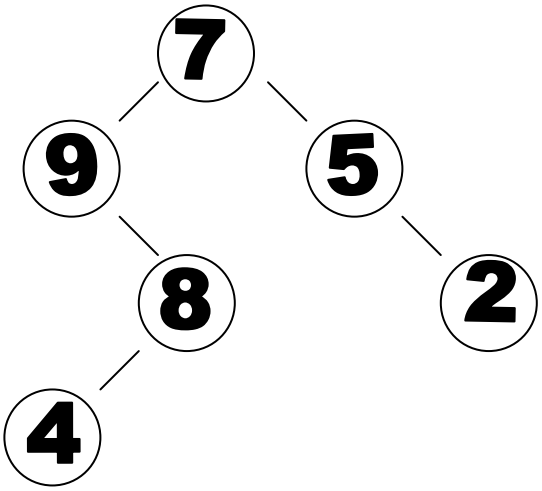
Dentre os vários tipos de árvores de busca, as mais simples são as árvores binárias de busca que veremos a seguir.

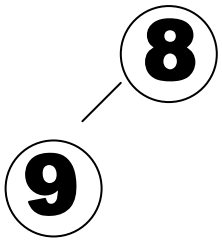
1. Árvores binárias

Chamamos de Árvores Binárias (AB), um conjunto finito `T` de nós ou vértices, onde existe um nó especial chamado **raiz** e os restantes podem ser divididos em dois subconjuntos disjuntos, chamados de sub-árvores esquerda e direita que também são Árvores Binárias. Em particular `T` pode ser vazio.

Exemplos:







Cada nó numa AB pode ter então 0, 1 ou 2 filhos. Portanto, existe uma hierarquia entre os nós. Com exceção da raiz, todo nó tem um nó pai.

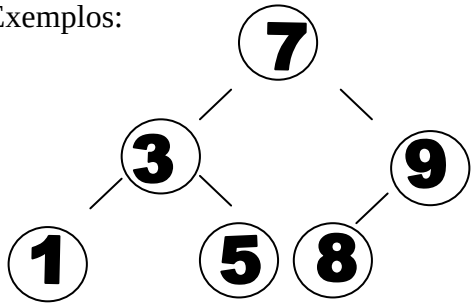
Dizemos que o nível da raiz é 1 e que o nível de um nó é o nível de seu pai mais 1. A altura de uma AB é o maior dos níveis de seus nós.

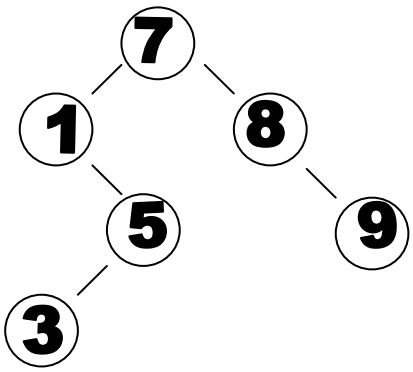
Dizemos que um nó é folha da AB se não tem filhos.

2. Árvores binárias de busca

Seja `T` uma AB. Se `v` é um nó de `T`, chamamos de `info(v)` a informação armazenada em `v`.
Chamamos `T` de Árvore Binária de Busca (ABB) quando:
Se `v1` pertencente à sub-árvore esquerda de `v` então `info(v1) < info(v)`.
Se `v2` pertencente à sub-árvore direita de `v` então `info(v2) > info(v)`.

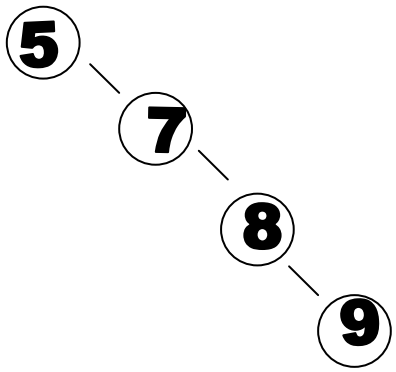
Exemplos:



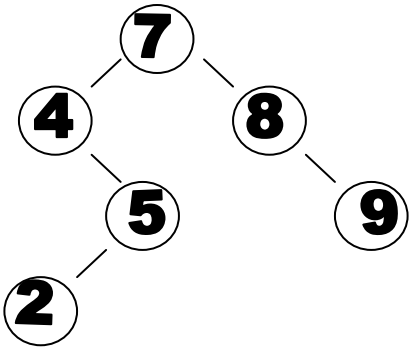


Os exemplos acima mostram que podemos ter várias ABB com os mesmos elementos. Conforme veremos à frente o objetivo é sempre termos uma ABB de menor altura. Nesse sentido a primeira ABB é melhor que a segunda.

Um exemplo de AB de muitos níveis e poucos elementos:



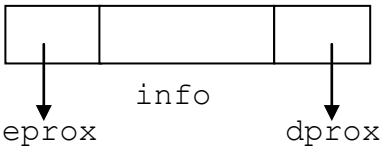
O exemplo abaixo não é ABB. O 2 está à direita do 4.



Uma ABB pode ter elementos repetidos. Podemos colocá-los na sub-árvore esquerda ou direita. Nos algoritmos abaixo vamos considerá-los sempre à direita. Dessa forma, os algoritmos para procurar um determinado elemento caso ele apareça mais vezes ficam mais simples.

3. Árvores binárias como listas ligadas

Podemos representar uma ABB com uma lista ligada, onde cada elemento tem os seguintes campos:



`info` - campo de informação
`eprox` - apontador para a sub-árvore esquerda
`dprox` - apontador para a sub-árvore direita

Como simplificação, vamos supor que o campo de `info` é um `int` (lembre-se que pode ser qualquer coisa: várias variáveis simples, vetores, `structs`, etc...) e definir as `structs` correspondentes:

```
struct item {
  int info;
  struct item * eprox, * dprox;
}
struct item raiz;
```

Uma outra forma com `typedef`:

```
typedef struct item * link;
struct item {
   int info;
   link eprox, dprox;
}
link raiz;
```

4. Algoritmos de busca

A1
Função que procura um determinado elemento numa ABB.
Chamada: `k = busca(raiz, x)`. Retorna `x` se encontrou elemento com `info` igual a `x`, ou `-1` se não encontrou.

```
int busca(link h, int v) {
   int t;
   if (h == NULL) return -1;
   t = h -> info;
   if (t == v) return t;
```

```
   if (v < t) return busca(h -> eprox), v);
   else return busca(h -> dprox, v)
}
```

A2
Outra versão retornando ponteiro para o elemento encontrado ou NULL se não encontrou.

```
link busca(link h, int v) {
   int t;
   if (h == NULL) return NULL;
   t = h -> info;
   if (v == t) return h;
   if (v < t) return busca(h -> eprox, v);
   else return busca(h -> dprox, v);
}
```

Complexidade da busca

No pior caso, o número de comparações é igual ao número de nós da árvore, no caso em que os elementos da árvore formam uma lista ligada num só sentido. Portanto a complexidade é O(N).

A complexidade é a altura da árvore, portanto é conveniente que a árvore tenha sempre altura mínima.
A árvore que possui tal propriedade é uma AB dita **completa** (todos os nós com filhos vazios estão no último ou penúltimo nível). Neste caso a complexidade é O(log N) ou seja: Se `T` é uma AB completa com `n>0` nós então `T` possui altura `h` mínima e $h = 1+\log_2 n$ (considerando o valor de $\log_2 n$ truncado).

O lema a seguir dá a relação entre altura e número de nós de uma AB completa:

Lema:
Seja `T` uma AB completa com `N` nós e altura `h`.
Então $2^{(h-1)} \leq N \leq 2^h - 1$.

Prova:
Se a AB completa possui apenas `1` nó no seu nível inferior então $N = 2^{(h-1)}$.
Se a AB completa está cheia $N = 2^h - 1$.

A3
Vejamos agora a versão não recursiva para a busca. A chamada `buscaNR(raiz, x)` procura elemento com `info` igual a `x` devolvendo um ponteiro para o mesmo ou `NULL` caso não encontre:

```
link buscaNR(link h, int v){
  link p; int t;
```

```
  p = h;
  while (p != NULL) {
     t = p->info;
     if (v == t) return p;
     if (v < t) p = p->eprox;
     else p = p->dprox;
  }
  return NULL;
}
```

5. Outros algoritmos

A4

A função a seguir conta o número de nós de uma AB com determinado valor de `info`. A chamada `conta(raiz, x)` devolve o número de elementos iguais a `x` da AB apontada por `raiz`.

```
int conta(link h, int c) {
   int a;
   if (h == NULL) return 0;
   if (c == h->info) a = 1 else a = 0;
   return a + conta(h->eprox,c) + conta(h->dprox,c);
}
```

Estamos supondo neste caso que os elementos iguais podem estar à direita ou à esquerda.
O algoritmo acima percorre toda a AB.
Refaça, supondo que se houver elementos iguais, estarão à direita.

A5

Transformar um vetor de `n` elementos, já ordenado, numa ABB mais ou menos equilibrada. A idéia é sempre pegar um elemento médio como raiz da sub-árvore. Para facilitar as chamadas recursivas vamos fazer a função de modo que a mesma se aplique a qualquer trecho contíguo do vetor. Assim, a chamada `raiz = monta(a, 0, n-1)` faz a montagem da árvore com os elementos `a[0]` até `a[n-1]`, devolvendo um ponteiro para a raiz da árvore. A chamada `raiz = monta(a, n1, n2)` faz o mesmo para os elementos `a[n1]` até `a[n2]`.

```
link monta(int a[], int left, int right) {
   int m = (left+right)/2; /* escolhe elemento médio */
   link h;
   if (left > right) return NULL;  /* sem elementos */
   /* insere o novo elemento */
   h = (link) malloc (sizeof(struct item));
   h->info = a[m];
   /* preenche os ponteiros */
   h->eprox = monta(a, left, m-1);
```

```
   h->dprox = monta(a, m+1, right);
   return h;
}
```

A6

Função que conta o número de nós de uma AB. A chamada `conta(raiz)`, devolve o número de nós da AB apontada por `raiz`.

```
int contaNN(link h) {
   if (h == NULL) return 0;
   return 1 + contaNN(h->eprox) + contaNN(h->dprox);
}
```

Exercícios

Baseado na solução acima escreva as seguintes funções:

1. Função `conta1(link h)` que conta o número de folhas de uma AB cuja raiz é h.
2. Função `conta2(link h)` que conta o número de nós com pelo menos um filho de uma AB cuja raiz é h.
3. Função `conta3(link h,int x)` que conta número de elementos com `info>= x` de uma ABB cuja raiz é `h`.
4. Idem ao problema A4 acima, considerando uma ABB onde elementos iguais ficam à direita.

6. Algoritmos de inserção numa ABB

Um novo elemento é inserido sempre como uma folha de uma ABB. É necessário descer na ABB até encontrar o nó que será o pai deste novo nó.

A6

Uma versão não recursiva para a inserção numa ABB. Supondo raiz como uma variável global.

```
void insere(int x) {
   link p, q;
   int z;
   /* verifica árvore vazia */
   if (raiz == NULL)
     {raiz = new(x, NULL, NULL); return;}
   /* procurar lugar e inserir */
   p = raiz; q = p;
   while (q != NULL) {
     z = q->info;
     if (x < z) {p = q; q = q->eprox;}
     else {p = q; q = q->dprox;}
   }
```

```
   /* p é o pai do nó a ser inserido, mas temos que verificar
      novamente se insere a esquerda ou direita de p */
   q = new(x, NULL, NULL);
   if (x < p->info) p->eprox = q;
   else p->dprox = q;
   return;
}

link new(int x, link left, link right) {
   /* cria novo nó com info x e links left e right */
   link q;
   q = (link) malloc (sizeof(struct item));
   q->info = x; q->eprox = left; q->dprox = right;
   return q;
}
```

Observe que se o elemento já estiver na ABB, será inserido na parte direita.

A6.0

Outra versão bem parecida com a anterior, mas sem usar dois ponteiros p e q para percorrer a ABB.

```
void insere(int x) {
   link p;
   int z;
   /* verifica árvore vazia */
   if (raiz == NULL)
     {raiz = new(x, NULL, NULL); return;}
   /* procurar lugar e inserir */
   p = raiz;
   while (1) {
     z = p -> info;
     /* verifica se insere a esquerda */
     if (x < z)
        if (p -> eprox == NULL) {
           p -> eprox = new(x, NULL, NULL); return;
        }
        else p = p -> eprox; /* continua a busca */
     else /* tentar inserir a direita */
          if (p -> dprox == NULL) {
             p -> dprox = new(x, NULL, NULL); return;
          }
          else p = p -> dprox;
   }
}
```

A6.1

Vejamos uma variação da função anterior usando ponteiro para ponteiro:

```
void insere(int x) {
   link p, *t;
   int z;
   /* verifica árvore vazia */
   if (raiz == NULL)
     {raiz = new(x, NULL, NULL); return;}
   /* procurar lugar e inserir */
   p = raiz;
   while (p != NULL) {
     z = p->info;
     if (x < z) {t = &(p->eprox); p = p->eprox;}
     else {t = &(p->dprox); p = p->dprox;}
   }
   /* t é apontador para o pai do nó a ser inserido */
   *t = new(x, NULL, NULL);
   return;
 }
```

A6.2

Outra variação, supondo agora que a raiz da ABB seja um parâmetro de entrada e saída da função. Note que a raiz pode ser alterada pela função quando a mesma é vazia, por isso o parâmetro tem que vir por endereço:

```
void insere(link *r, int x) {
   link p, q;
   int z;
   /* verifica árvore vazia */
   if (*r == NULL)
     {*r = new(x, NULL, NULL); return;}
   /* procurar lugar e inserir */
   p = *r; q = p;
   while (q != NULL) {
     z = q->info;
     if (x < z) {p = q; q = q->eprox;}
     else {p = q; q = q->dprox;}
   }
   /* p é o pai do nó a ser inserido */
   q = new(x, NULL, NULL);
   if (x < p->info) p->eprox = q;
   else p->dprox = q;
   return;
}
```

A6.3

A versão recursiva abaixo devolve a cada chamada, o próprio nó se diferente de `NULL` ou um apontador para um novo nó que será inserido. A chamada `raiz = insere(raiz, x)` insere elemento com `info` igual a `x` na ABB apontada por `raiz`. A atribuição à `raiz` é porque a árvore pode estar vazia.

```
link insere(link h, int x) {
   int z;
   /* verifica árvore vazia */
   if (h == NULL) return new(x, NULL, NULL);
   /* procurar lugar e inserir */
   z = h->info;
   if (x < z) h->eprox = insere(h->eprox, x)
   else h->dprox = insere(h->dprox, x)
   /* devolve o próprio nó para não alterar os ponteiros */
   return h;
}
```

Complexidade da construção de uma ABB por inserções sucessivas

Para inserir elemento é necessário achar o seu lugar. Portanto a complexidade é a mesma da busca.

Usando-se o algoritmo acima e inserindo-se um a um, podemos no pior caso (ABB com um só elemento por nível - tudo à esquerda, tudo à direita ou ziguezague) chegar a: `1+2+3+...+n = n.(n+1)/2` acessos para construir toda a árvore. Portanto $O(n^2)$.

Se os elementos a serem inseridos estiverem ordenados, usando o algoritmo A5, a complexidade é O(N). Mas é necessário ordenar antes.

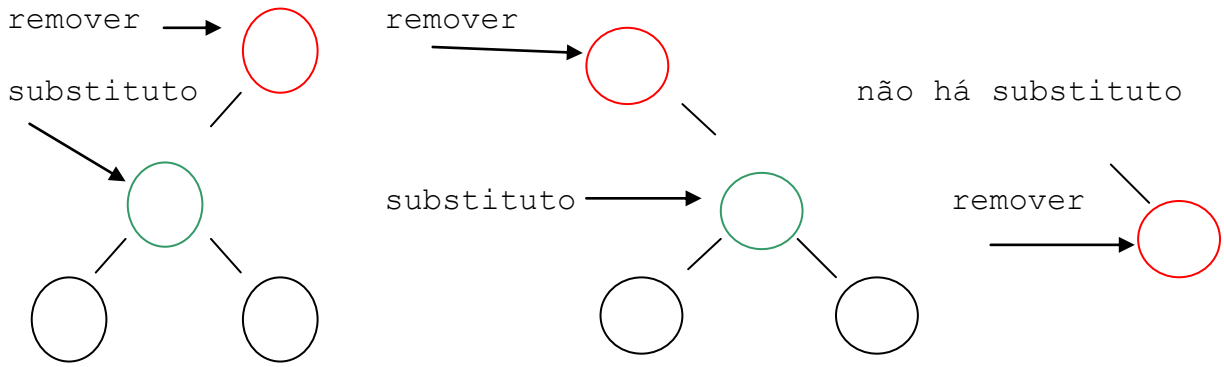
Para inserir um elemento, supondo a árvore completa (folha da árvore) teremos que percorrer os níveis que serão `1+log n`. Portanto temos um algoritmo `O(log n).`

7. Algoritmo de remoção numa ABB

A remoção é um pouco mais complexa que a busca ou inserção. O problema da remoção física de um nó é que é necessário encontrar outro nó para substituir o removido, caso o nó a ser removido tenha filhos.

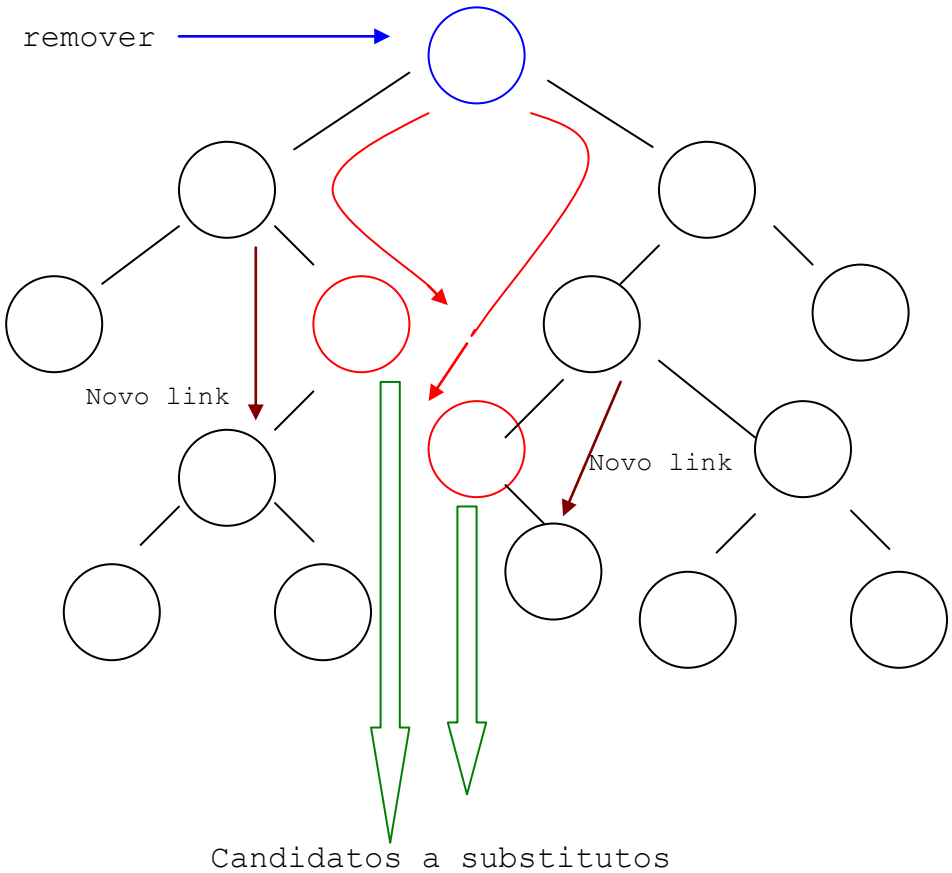
Dois casos a considerar:

1) O nó a ser removido não tem filhos esquerdo e/ou direito.



É só alterar o ponteiro para o nó a substituir e remover fisicamente o nó. Se não há filhos, basta mudar o ponteiro do pai para `NULL`.

2) O nó a ser removido tem filhos direito e esquerdo:



Os candidatos a substituto são obtidos percorrendo-se a ABB:
Um à esquerda e tudo a direita até achar nó com `dprox NULL` . Ou um a direita e tudo à esquerda até achar nó com `eprox NULL`.
Além de alterar o ponteiro para o nó que vais substituir, é necessário mover o conteúdo deste nó para o nó a remover e fisicamente remover o substituto. O pai do substituto assume os seus filhos.

Nos algoritmos de remoção, vamos usar ponteiros para ponteiros. Só recordando, considere a declaração:

```
link *pp;
```

`**pp` é do tipo `struct item`
`*pp` é do tipo ponteiro para `struct item`
`pp` é do tipo ponteiro para ponteiro para `struct item`

A7
O primeiro passo é procurar o nó a remover. Em seguida verificar os dois casos:

A função abaixo procura nó com `info x`, devolvendo ponteiro para o ponteiro deste nó, isto é, devolvendo o ponteiro para o ponteiro que será alterado para eliminar este elemento:

```
link *search(link *r, int x) {
   link *q;
   q = r; /* inicia q com a raiz */
   /* procura na ABB */
   while (*q != NULL) {
     if ((*q)->info == x) return q;
     /* esquerda ou direita */
     if (x < (*q)->info) q = &((*q)->eprox);
     else q = &((*q)->dprox)
   }
   /* se chegou aqui é porque não encontrou o x e q aponta
      para um ponteiro que é NULL ou ainda para um ponteiro
      aonde será inserido um elemento */
   return q;
}
```

A8
Vamos agora à remoção usando `search` acima. A função abaixo remove um nó cujo ponteiro é apontado por `*pp`.

```
void delnode(link *pp) {
   link p, *qq, q;
   /* se *pp é NULL nada a fazer */
   if (*pp == NULL) return;
```

```
   /* verifica qual o caso - sem filho esquerdo ou direito */
   p = *pp;
   if (p->dprox) == NULL) {
      /* muda o pai e libera */
      *pp = p->eprox;
      free(p);
   }
   else if (p->eprox == NULL) {
           /* muda pai e libera */
           *pp = p->dprox;
           free(p);
         }
        else {/* um para esquerda e tudo à direita */
              qq = &(p->eprox);
              /* procura primeiro dprox NULL */
              while ((*qq)->dprox != NULL)
                 qq = &((*qq)->dprox;
              /* achamos o substituto */
              q = *qq;
              /* altera ponteiro do pai de q */
              *qq = q->eprox;
              /* move as info */
              p->info = q->info;
              free(q); // libera o tal nó
              return;
        }
}
```

Para eliminar um nó fazemos a seguinte seqüência:

```
link *t;
...
/* elimina nó com info igual a x */
t = search(&raiz, x);
delnode(t);

/* outra forma */
delnode(search(&raiz, x));
```

Complexidade da remoção

O pior caso é quando a árvore tem um só elemento por nível.
Como `search` é O(n) e `delnode` é O(n), o total é O(n).

A9

Outra solução para a inserção usando `search`. A chamada seria `insert(&raiz, x)`.

```
void insert(link *r, int x) {
   link *qq;
   qq = search(r, x);
   if (*qq == NULL) {
      /* não encontrou então pode inserir */
      /* note que qq aponta para o pai */
      *qq = (link) malloc (sizeof(struct item));
      (*qq)->info = x;
      (*qq)->eprox = (*qq)->dprox = NULL;
   }
}
```

8. Árvores Binárias de Busca Completas

Já vimos que o problema das ABB é que ela pode ficar desbalanceada com a inserção e remoção de novos elementos. A situação ideal em uma ABB é que ela se já **completa** (como o menor número possível de níveis).
Como seria possível mantê-la **completa**?
Isso pode ser feito de 2 maneiras:

1) Toda vez que um elemento é inserido ou removido, rearranja-se a ABB para a mesma continue **completa**.
2) Inserir e remover elementos da maneira usual e de tempos em tempos executar um algoritmo que reconstrói a ABB deixando-a **completa**.

Existem vários algoritmos com esses objetivos. **Não serão vistos neste curso**.
Apenas citamos 2 tipos mais comuns abaixo. Nessas ABBs, os algoritmos de inserção e remoção já o fazem deixando a ABB completa ou balanceada.

Com uma ABB **completa**, chegamos a situação ideal de busca, pois temos uma algoritmo equivalente ao da busca binária O(log N), em uma tabela que permite inserções rápidas (O(log N)) e remoções tão rápidas quanto possível (O(N) no pior caso). Além disso, só usa a quantidade de memória necessária.

9. Outras Árvores Binárias

Apenas citando os tipos mais importantes:

9.1 Árvores Binárias de Busca AVL (Adelson-Vesky e Landis (1962))

Cada nó mantém uma informação adicional, chamada fator de balanceamento que indica a diferença de altura entre as sub-árvores esquerda e direita.
As operações de inserção e remoção mantém o fator de balanceamento entre -1 e +1.

9.2 Árvores Binárias de Busca Rubro-Negras

É uma ABB com as seguintes propriedades:

1. Todo nó é vermelho ou preto.
2. Toda folha é preta.
3. Se um nó é vermelho então seus filhos são pretos.
4. Todo caminho da raiz até qualquer folha tem sempre o mesmo número de nós pretos.

Com essas propriedades, é possível manter a ABB mais ou menos balanceada após inserções e remoções.

10. Outras Árvores de Busca

Árvores de Busca, não precisam ser necessariamente binárias. Podemos construir árvores com vários elementos em cada nó (n-árias). Cada elemento possui um ramo esquerdo (menores) e um ramo direito (maiores ou iguais).
Este é o caso das chamadas B-Árvores. Também **não serão vistos neste curso.**
São usadas principalmente para arquivos em banco de dados.
No caso de arquivos interessa muito diminuir a quantidade de acessos a disco. Assim, a cada leitura, vários nós estarão disponíveis na memória. A quantidade de níveis da árvore diminui e, portanto a quantidade de acessos para se procurar um elemento.