O uso de Árvore Binária de Busca em tabela de Símbolos pode trazer vantagens relacionadas a outras abordagens vistas anteriormente.

Definição Uma Árvore Binária de Busca (BST) é uma árvore binária que possui uma CHAVE (Key) associada a cada um dos nós internos, com a propriedade adicional de que a chave de qualquer nó é maior (ou igual) que as chaves de todos os nós na sub-árvore à esquerda e é menor (ou igual) que as chaves de todos os nós na sub-árvore à direita.

Implementação básica utiliza métodos recursivos de Inserção e busca. Define-se um tipo "link" que carrega um Item, links para sub-árvores da esquerda e direita e uma contagem de nós nos sub-árvores +1.

- STinsert/insertR - STsearch/searchR STcount STinit/NEW

```
typedef struct STNode * link;
struct stNode { Item item; link l, r; int N; };
link h, z;
link NEW(Item item, link l, link r, int N)
{ link x = malloc(sizeof(*x));
  x->Item = item; x->l = l; x->r = r; x->N = N;
  return x;
}
Void STinit( )
{ h = (z = NEW(NULLItem, 0, 0, 0)); }
int STcount( )
{ return h->N; }
```

- A função "sort" para a tabela de símbolo é facilmente implementada, adicionando um pequeno trabalho
  - A árvore binária de busca já representa um arquivo ordenado se olhamos do jeito correto. Uma visita <u>inorder</u> resolve o problema.

```
Void sortR(link r, void *(visit)(Item))
{
   if(r==z) return;
   sortR(r->l, visit);
   visit(r->item); //"U"
   sortR(r->r, visit);
}
Void STsort(void *(visit)(Item))
{
   sortR(h, visit);
}
```

```
Item v[10000];
void trabalho(Item it)
{
  static int i=0;
  v[i++]=it;
}
```

```
stsort(trabalho);
```

→ Performance de uma BST

propriedade Uma busca bem sucedida requer $\sim 1{,}39 \lg N$ comparações, na média, em uma BST construída com N chaves aleatórias.

→ O número de comparações usado em uma busca bem sucedida é 1 + (a distância do nó à raiz).

→ Tomando a distância para todos os nós, temos o caminho interno da árvore. Logo,

1 + (média dos caminhos internos de BST)

Seja C_N a média do comprimento dos caminhos internos de uma BST de N nós, temos a recorrência

$$C_N = N - 1 + \frac{1}{N} \cdot \sum_{1 \le k \le N} (C_{k-1} + C_{N-k})$$

$C_1 = 1 \cdot (N-1)$ é o quando a raiz contribui para o comprimento dos caminhos internos (sendo 1)

Assumindo que a raiz é, provavelmente, o k-ésimo menor elemento, deixando duas sub-árvores de tamanho $k-1$ e $N-k$

propriedade Inserções e buscas mal sucedidas requerem $\sim 1{,}39 \lg N$ comparações, na média, em uma BST construída com N chaves aleatórias.

→ A busca por uma chave aleatória qualquer provavelmente termina em algum dos $N+1$ nós externos em uma busca mal sucedida.

Esta propriedade é pareada com o fato de que a diferença entre o comprimento de caminho externo e comprimento de caminho interno é $2 \cdot N$

lembrando:

propriedade: Uma árvore binária com N nós internos tem 2N links: N-1 links para nós internos
N+1 links para nós externos

definições: - level de um nó é 1+altura do level do pai
- altura: é o máximo de "levels" dos nós
- Comprimento de caminho: soma do todos os levels dos nós internos
- Comprimento de caminho interno: Soma dos levels dos nós internos
- Comprimento de caminho externo: Soma dos levels dos nós externos

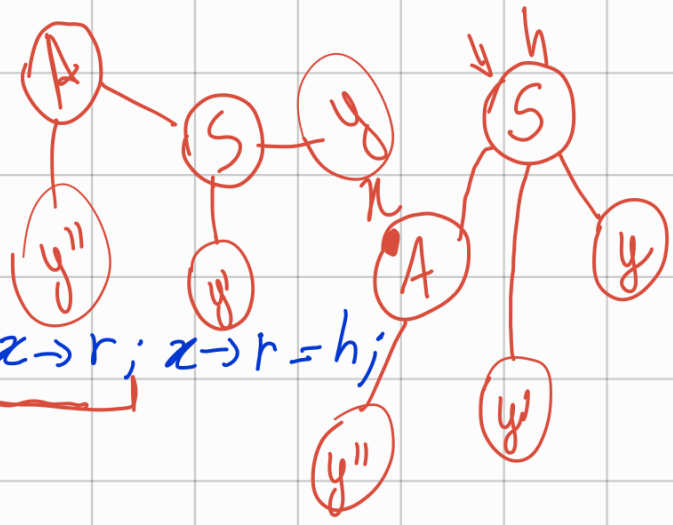
propriedade: O comprimento de caminho externo de uma árvore com N nós internos é 2N maior que o comprimento de caminho interno

---

Propriedade No pior caso, uma busca em uma BST com N chaves pode custar N comparações

Inserção na Raiz em BST

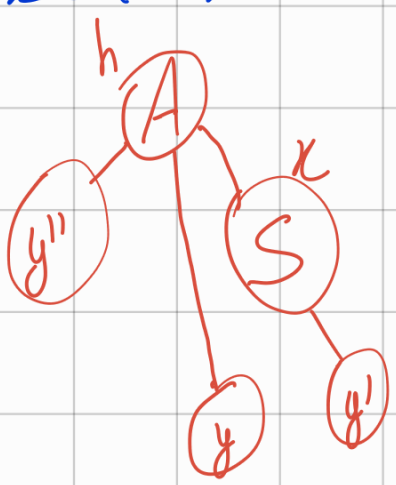
- Deixar a chave recém-inserida na raiz.
  - Exige deixar a árvore saudável (respeitando BST)



Rotações

```
link rotR(link h)
{ link x = h->l; h->l = x->r; x->r = h;
  return x;
}
```

```
link rotL(link h)
{ link x = h->r; h->r = x->l; x->l = h;
  return x;
}
```



→ Como fazer a inserção?

```
link InsertT(link h, Item item)
{
  if(h==r) return NEW(item, z, z, 1);
  Key k = key(item), t = key(h->item);
  if(less(k,t)) { h->l = insertT(h->l, item); h = rotR(h); }
  else          { h->r = insertT(h->r, item); h = rotL(h); }

  return h;
```

→ Seleção em BST

→ Buscar o K-ésimo menor elemento da BST.

O algoritmo é simples: Verifica o número de nós na sub-árvore da esquerda.

- Se há K nós - retorna o item de raiz
- Senão, se a árvore da esquerda tem mais de K-nós, recursivamente ovenue para o item e esquerda.
- Se nenhuma das condições são verdadeiras, então a sub-árvore da esquerda tem "t" items, com $t < K$, e o K-ésimo menor item na BST é o item $(k-t-1)$-ésimo menor da árvore da direita.

```
link InsertR(link r, Item item)
{
  if(r == z) return NEW(item, y, z, 1);
  Key k = Key(item), t = Key(r->item);
  if(less(k, t)) r->l = insertR(r->l, item);
  else           r->r = insertR(r->r, item);
  (r->N)++;
  return r;
}

Void STinsert(Item item)
{
  h = insertR(h, item);
}

Item searchR(link r, Key k)
{
  if(r == z) return NULLitem;
  Key t = Key(r->item);
  if(eq(k, t)) return r->item;
  if(less(k, t)) return searchR(r->l, k);
  return searchR(r->r, k);
}

Item STsearch(Key k)
{
  return searchR(h, k);
}
```