Algoritmos de Busca

Profª. Rose Yuri Shimizu

Roteiro

1. Tabela de Símbolos

2. Algoritmos de Busca
   - Busca binária
   - Árvore binária de busca

Tabela de Símbolos

- Coleção de pares de chave-valor
- Mecanismo abstrato para armazenar informações que podem ser acessadas através de uma chave
- Algumas aplicações

Aplicação	Chave	Valor
dicionário	palavra	definição
índice de livro	termo	lista de páginas de números
compartilhamento de arquivos	nome do arquivo	seed
sistemas de transações	número da conta	detalhes da transação
compilador	nome da variável	tipo e valor
sistemas de reservas	reserva	cliente+vôo

Tabela de Símbolos

- Chaves duplicadas: em muitas aplicações não são permitidas
- Dificuldade:
  - Definir uma estrutura que represente uma tabela de símbolos capaz de armazenar uma grande quantidade de dados (informações + chaves)
  - Definir formar de recuperar essas informações eficientemente
- Exemplo de tabelas disponíveis nas linguagens: dicionários, map

Tabela de Símbolos

- Propósito:
  - Associação de chave-valor
  - A partir de uma chave, achar o valor associado
- Computacionalmente:
  - Estrutura de dados + algoritmo de busca
  - Operações básicas: inserção, busca e remoção

Tabela de Símbolos - Estruturas de dados

- Vetores com chaves indexadas
  - Índices são (ou representam) as chaves
  - Conteúdo são os valores associados as chaves
  - Sem chaves duplicadas
  - Inserção de item com chave pré-existente: sobrescreve
  - Chaves: intervalo pequenos
  - Algoritmo de busca: acesso direto
- Vetores ordenados
- Árvores binárias

Tabela de Símbolos - Estruturas de dados

- Vetores com chaves indexadas
- Vetores ordenados
  - Conteúdo composto por chave+valor
  - Admite chaves duplicadas
  - Chaves: intervalo grandes
  - Algoritmo de busca: sequencial, binário
- Árvores binárias

Tabela de Símbolos - Estruturas de dados

- Vetores com chaves indexadas
- Vetores ordenados
- Árvores binárias
  - Algumas permitem chaves duplicadas
    - ⋆ Buscas e remoções: seguem a ordem de inserção
  - Intervalos grandes
  - Algoritmo de busca: profundidade (Depth First Search – DFS), largura (Breadth-First Search - BFS), binária
  - heap, priority queue, red-black, avl ...

Roteiro

Roteiro

1. Tabela de Símbolos

2. Algoritmos de Busca
   - Busca binária
   - Árvore binária de busca

Busca binária em vetores ordenados

- Paradigma da divisão e conquista
  - Dividir o vetor no meio
  - Procurar o elemento na esquerda: elemento procurado seja menor que o elemento central
  - Procurar o elemento na direita: elemento procurado seja maior que o elemento central
  - Repetir, recursivamente, até o elemento procurado ser o elemento central (ou não - falha na busca)

Busca binária em vetores ordenados

```
1   #define key(A) (A.chave)
2
3   typedef int Key;
4   typedef struct data Item;
5   struct data { Key chave; char info[100]; };
6
7   Item binary_search(Item *v, int l, int r, Key k)
8   {
9       if(l >= r) return NULL;
10
11      int m = (l+r)/2; //l+(r-l)/2
12      if(k == key(v[m])) return v[m];
13      if(k < key(v[m]))
14          return binary_search(v, l, m-1, k);
15
16      return binary_search(v, m+1, r, k);
17  }
18
```

- Complexidade: até $\lfloor lgN \rfloor + 1$ comparações (acerto ou falha)
- https://fga.rysh.com.br/eda1/pdf/3-complexidade.pdf

Busca binária em vetores ordenados

- *Interpolation search*
- Mais próximo do início ou fim do vetor
  - $l + (r - l) * 1/2$
    - ⋆ $1/2$: posição do elemento do meio
    - ⋆ $(k - inicio)/(fim - inicio)$:
      - proporção dos k elementos iniciais em relação ao total
  - $l + (r - l) * ((k - key(v[l]))/(key(v[r]) - (key(v[l])))$
- Exemplo
  - 2 3 5 8 10 21 32 ; k = 21
  - intervalo total (32-2) = 30
  - intervalo da chave (21-2) = 19
  - proporção $19/30 \approx 0.63$
  - começa a procurar mais próximo do fim

Busca binária em vetores ordenados

Interpolation search

```
1  Item binary_search(Item *v, int l, int r, Key k)
2  {
3      if(l >= r) return NULL;
4
5      int m = l + (r-l)*((k-key(v[l]))/(key(v[r])-(key(v[l]))));
6
7      if(k == key(v[m])) return v[m];
8      if(k < key(v[m]))
9          return binary_search(v, l, m-1, k);
10
11     return binary_search(v, m+1, r, k);
12 }
```

- Interessante para muitas chaves, mas ...
- é altamente dependente da boa distribuição das chaves

Busca binária em vetores ordenados

Interpolation search

```
1  Item binary_search(Item *v, int l, int r, Key k)
2  {
3      if(l >= r) return NULL;
4
5      int m = l + (r-l)*((k-key(v[l]))/(key(v[r])-(key(v[l]))));
6
7      if(k == key(v[m])) return v[m];
8      if(k < key(v[m]))
9          return binary_search(v, l, m-1, k);
10
11     return binary_search(v, m+1, r, k);
12 }
```

- Interessante para muitas chaves, mas ...
- é altamente dependente da boa distribuição das chaves
  - 1 2 3 4 5 21 90 ; k = 21
  - proporção 20/89 ≈ 0.22 : mais próximo do início

Busca binária em vetores ordenados

Interpolation search

```
1  Item binary_search(Item *v, int l, int r, Key k)
2  {
3      if(l >= r) return NULL;
4
5      int m = l + (r-l)*((k-key(v[l]))/(key(v[r])-(key(v[l]))));
6
7      if(k == key(v[m])) return v[m];
8      if(k < key(v[m]))
9          return binary_search(v, l, m-1, k);
10
11     return binary_search(v, m+1, r, k);
12 }
```

- Interessante para muitas chaves, mas ...
- é altamente dependente da boa distribuição das chaves
  - 1 2 3 4 5 21 90 ; k = 21
  - proporção $20/89 \approx 0.22$: mais próximo do início

Busca binária em vetores ordenados

Interpolation search

```
1  Item binary_search(Item *v, int l, int r, Key k)
2  {
3      if(l >= r) return NULL;
4
5      int m = l + (r-l)*((k-key(v[l]))/(key(v[r])-(key(v[l]))));
6
7      if(k == key(v[m])) return v[m];
8      if(k < key(v[m]))
9          return binary_search(v, l, m-1, k);
10
11     return binary_search(v, m+1, r, k);
12 }
```

- Interessante para muitas chaves, mas ...
- é altamente dependente da boa distribuição das chaves
  - 1 2 3 4 5 21 90 ; k = 21
  - proporção $20/89 \approx 0.22$: mais próximo do início

Roteiro

1. Tabela de Símbolos

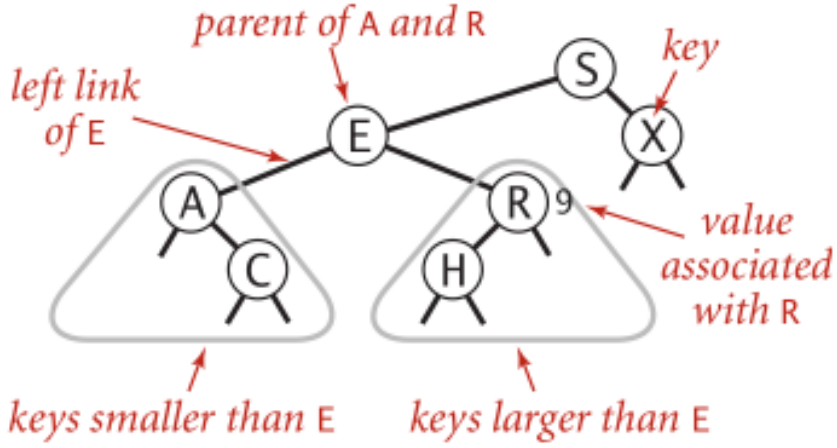
2. Algoritmos de Busca
   - Busca binária
   - Árvore binária de busca

Árvore binária de busca

- Combina a flexibilidade da inserção nas lista encadeadas com a eficiência da busca nos vetores ordenados
- Estrutura das árvores binárias
  - Todo nó interno tem dois ponteiros que apontam para filho à esquerda e à direita
  - Nó folha apontam para NULL

Árvore binária de busca

- Chaves: conteúdo dos nós
- Estrutura:
  - Cada nó tem a chave maior as chaves da sub-árvore esquerda
  - Cada nó tem a chave menor as chaves da sub-árvore direita
- Estrutura: permite a busca binária de um nó a partir da raiz



Anatomy of a binary search tree

Árvore binária de busca

Exemplo de estrutura básica

```
1  #define info(A) (A.info)
2  #define key(A) (A.chave)
3  #define less(A, B) ((A) < (B))
4  #define eq(A, B) ((A) == (B))
5  #define exch(A, B) { Item t=A; A=B; B=t; }
6  #define compexch(A, B) if(less(B, A)) exch(A, B)
7
8
9  typedef int Key;
10 typedef struct data Item;
11 struct data {
12     Key chave;
13     char info[100];
14 };
15
16 typedef struct node STnode;
17 struct node {
18     Item item;
19     STnode *esq;
20     STnode *dir;
21 };
```

Árvore binária de busca

```
22
23 STnode *new(Item x, STnode *e, STnode *d)
24 {
25     STnode *no = malloc(sizeof(STnode));
26     no->esq = e;
27     no->dir = d;
28     no->item = x;
29     return no;
30 }
```

Árvore binária de busca - Busca binária

- Como sua estrutura divide entre maiores e menores, a busca binária acontece a partir de sua raiz
- Caso o item procurado seja menor que a raiz, procure na sub-árvore esquerda
- Caso contrário, procure na sub-árvore direita

```
1  STnode *STsearch(STnode *no, Key v)
2  {
3      if(no == NULL || eq(v, key(no->item)))
4          return no;
5
6      if(less(v, key(no->item)))
7          return STsearch(no->esq, v);
8      else
9          return STsearch(no->dir, v);
10 }
```

Árvore binária de busca - Inserção

- As propriedades da árvore binária de busca devem ser mantidas
  - Elementos menores para esquerda
  - Elementos maiores para direita

Árvore binária de busca - Inserção

- As propriedades da árvore binária de busca devem ser mantidas
  - Elementos menores para esquerda
  - Elementos maiores para direita

Árvore binária de busca - Inserção

- As propriedades da árvore binária de busca devem ser mantidas
  - Elementos menores para esquerda
  - Elementos maiores para direita

Árvore binária de busca - Inserção

```
1  STnode *STinsert(STnode *no, Item item){
2
3      if(no == NULL)
4          return new(item, NULL, NULL);
5
6      Key novo = key(item);
7      Key atual = key(no->item);
8      if(less(novo, atual))
9          no->esq = STinsert(no->esq, item);
10     else
11         no->dir = STinsert(no->dir, item);
12
13     return no;
14 }
```

Árvore binária de busca - Remoção

- Se tiver filho único, este "assume" sua posição
- Se tiver dois filhos, outro nó "adota" seus filhos e "assume" sua posição
  - pai adotivo: tenha no máximo 1 filho que será adotado pelo avô

Árvore binária de busca - Remoção

- Como garantir que o novo nó seja menor que todos os elemento à direita do removido?
  - Sendo o menor dos maiores
  - Sendo o menor dos itens da sub-árvore direita
- Como garantir que o novo nó seja maior que todos os elemento à esquerda do removido?
  - Sendo o maior dos menores
  - Sendo o maior dos itens da sub-árvore esquerda

Árvore binária de busca - Remoção

- Sendo o nó reposicionado o menor dos maiores ou o maior dos menores, garante-se as propriedades pois os elementos das sub-árvores:
  - esquerda $<$ direita
  - esquerda $<$ menor da direita $<$ direita
  - esquerda $<$ maior da esquerda $<$ direita

Árvore binária de busca - Remoção

```
1  //remove o primeiro nó com a chave "remove"
2  STnode *STdelete(STnode *no, Key remove)
3  {
4      //não achou
5      if (no == NULL) return NULL;
6
7
8      Key atual = key(no->item);
9
10
11     //procure à esquerda
12     if(less(remove, atual))
13         no->esq = STdelete(no->esq, remove);
14
15
16
17     //procure à direita
18     else if(less(atual, remove))
19         no->dir = STdelete(no->dir, remove);
20
21
22
```

Árvore binária de busca - Remoção

```
23      //achou
24      else
25      {
26          if (no->dir == NULL) {
27              //filho único: retorne o filho a esquerda
28              //  para rearranjar a árvore
29              STnode *aux = no->esq;
30              free(no);
31              return aux;
32          }
33
34
35          if (no->esq == NULL){
36              //filho único:  retorne o filho a direita
37              //  para rearranjar a árvore
38              STnode *aux = no->dir;
39              free(no);
40              return aux;
41          }
42
43
44
```

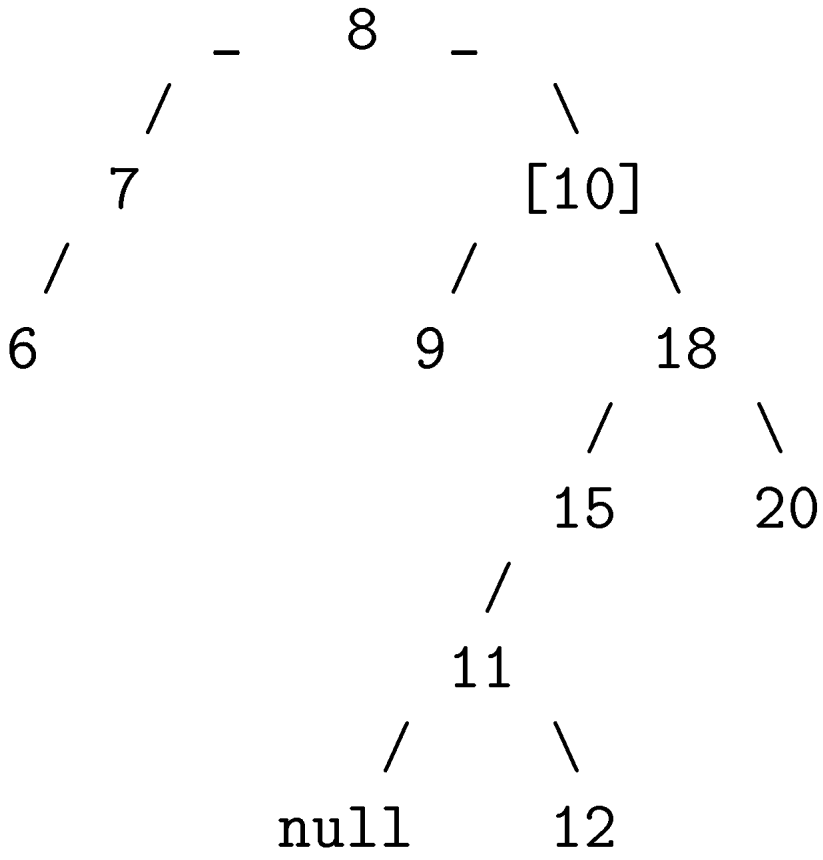
Árvore binária de busca - Remoção

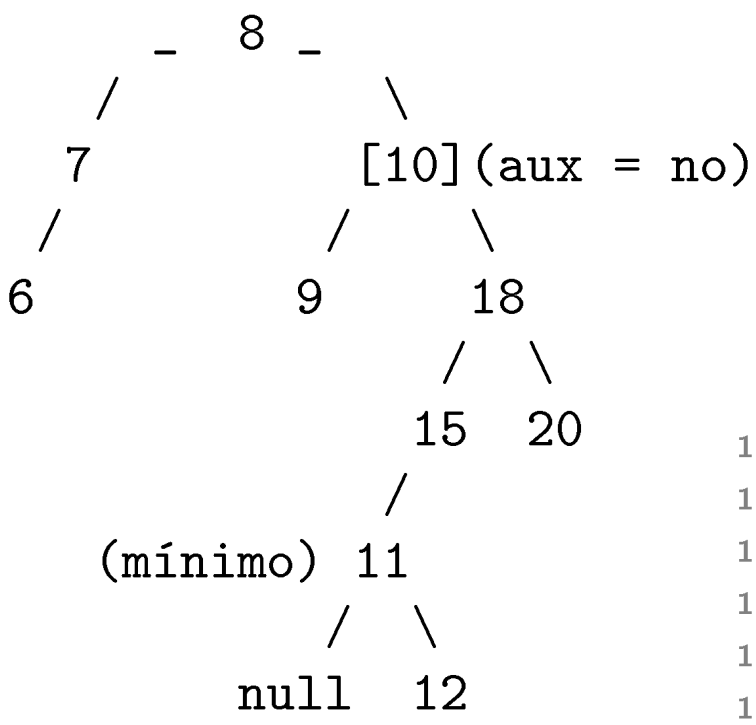
```
45          //Se tiver os dois filhos
46          //  achar um substituto
47          STnode *aux = no;
48
49          //menor dos maiores
50          no = min(aux->dir);
51
52          //remova o substituto e
53          //substituto assume filho direito
54          no->dir = deleteMin(aux->dir);
55
56          //substituto assume filho esquerdo
57          no->esq = aux->esq;
58
59          //libera memória do removido
60          free(aux);
61      }
62      return no;
63  }
64
65
66
```

Árvore binária de busca - Remoção

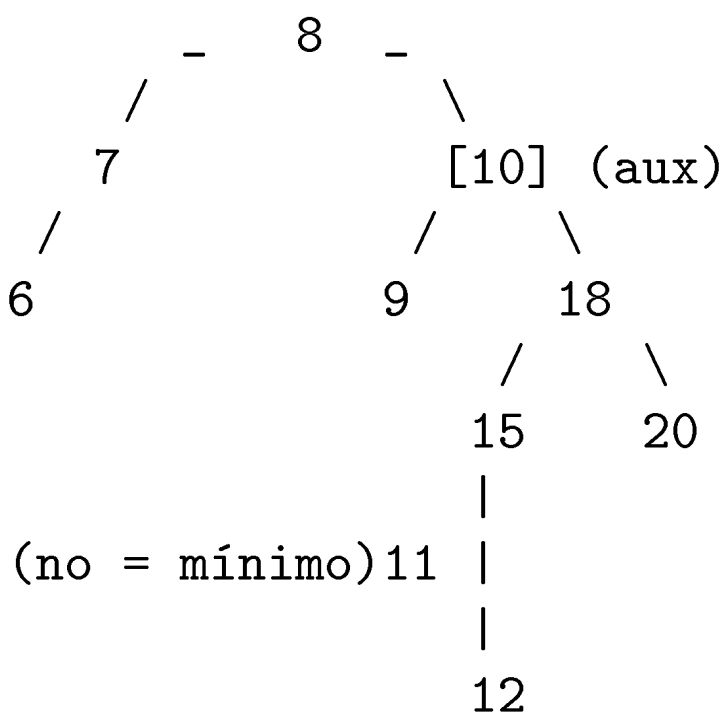
```
67 //menor = elemento mais à esquerda
68 STnode *min(STnode *no)
69 {
70     if (no->esq == NULL) return no;
71     return min(no->esq);
72 }
73
74 //"removendo" o menor elemento a partir de um nó
75 STnode *deleteMin(STnode *no)
76 {
77     //achando o menor, devolva o filho da direita para
78     //    atualizar a esquerda do pai do menor
79     if (no->esq == NULL) return no->dir;
80
81     no->esq = deleteMin(no->esq);
82     return no;
83 }
84
```

```
1 //Exemplo de execução
2 tree = STdelete(tree, 10);
```

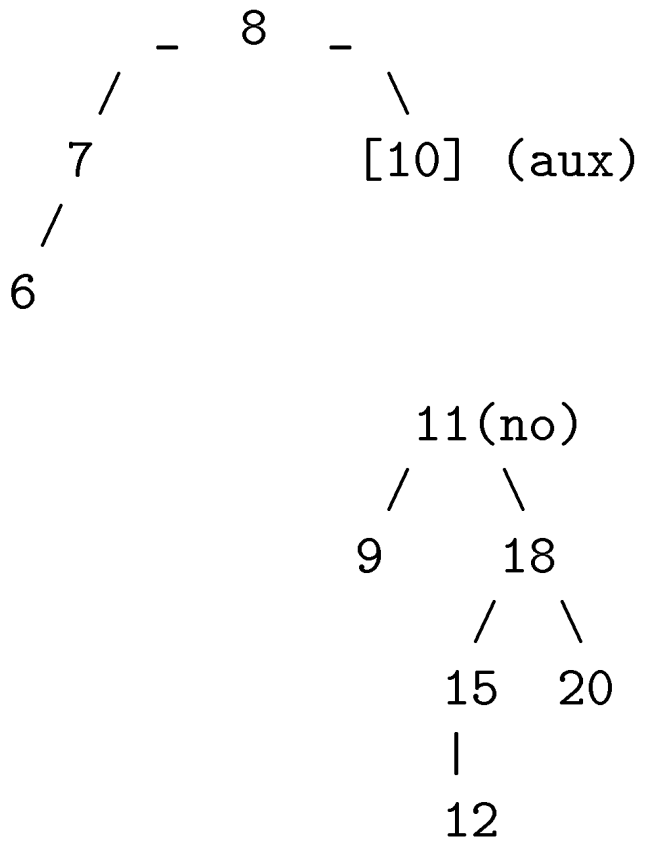




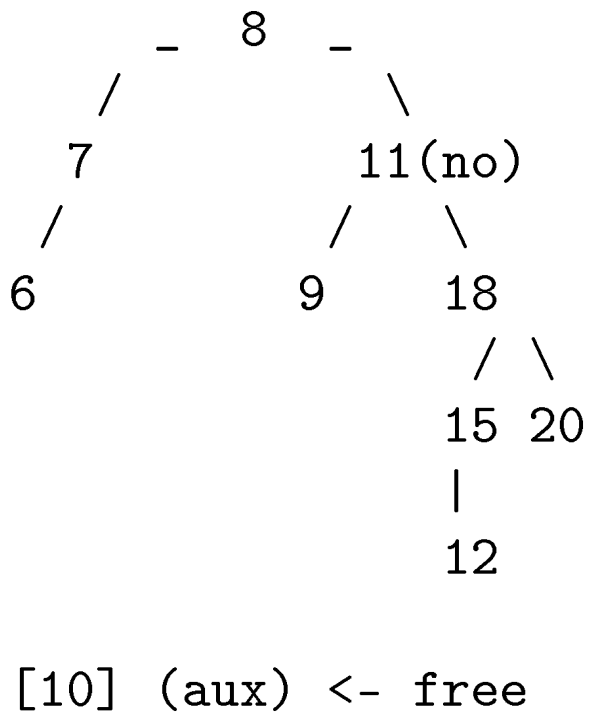

```
1  STnode *min(STnode *no) {
2    if (no->esq == NULL)
3          return no;
4
5    return min(no->esq);
6  }
7
8  STnode *STdelete(STnode *no,
9                   Key remove)
10 { ...
11
12   //achou
13   else {
14      ...
15     STnode *aux = no;
16     no = min(aux->dir); //
17     no->dir = deleteMin(aux->dir);
18     no->esq = aux->esq;
19     free(aux);
20   }
21   return no;
22 }
```




```
1  STnode *deleteMin(STnode *no) {
2    if(no->esq == NULL)
3          return no->dir;
4
5    no->esq = deleteMin(no->esq); //
6
7    return no;
8  }
9
10 STnode *STdelete(STnode *no,
11                    Key remove)
12 { ...
13
14   //achou
15   else {
16      ...
17     STnode *aux = no;
18     no = min(aux->dir);
19     no->dir = deleteMin(aux->dir);//
20     no->esq = aux->esq;
21     free(aux);
22   }
23   return no;
24 }
```




```
1  STnode *STdelete(STnode *no,
2                   Key remove)
3  { ...
4
5    //achou
6    else {
7      ...
8      STnode *aux = no;
9      no = min(aux->dir);
10     no->dir = deleteMin(aux->dir);//
11     no->esq = aux->esq;//
12     free(aux);
13   }
14   return no;
15 }
```




```
1  STnode *STdelete(STnode *no,
2                   Key remove)
3  { ...
4
5    //achou
6    else {
7      ...
8      STnode *aux = no;
9      no = min(aux->dir);
10     no->dir = deleteMin(aux->dir);
11     no->esq = aux->esq;
12     free(aux); //<-
13   }
14   return no; //<-
15 }
```

```
1  //Manipulação dos endereços
2  STnode *STdelete(STnode *no, Key remove) {
3      ...
4      //achou
5      else { ...
6          /*  memória
7                aux|      |          aux| &c1 |    free(aux)
8                no | &c1 |           no | &c4 |       |
9               ... |      |         ... |     |       |
10               c1 | 17   |          c1 | --  | <---
11              ... |      |         ... |     |
12               c4 | 11   |          c4 | 11  |
13         */
14         STnode *aux = no;
15         no = min(aux->dir);
16         no->dir = deleteMin(aux->dir);
17         no->esq = aux->esq;
18         free(aux);
19     }
20     return no;
21 }
```

Árvore binária de busca - Performance

- Depende do balanceamento da árvore
- Balanceamento:
  - Nós bem distribuídos nas sub-árvores
    - ⋆ Sub-árvores esquerda e direita $\approx$ mesma altura
  - Altura $\approx lgN$
  - Operações de rotações
- Exemplos de árvores balanceadas
  - Árvores AVL
  - **Árvores Red-Black** (rubro-negra/vermelha-preta)
  - `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`
- Melhor caso: lgN
- Pior caso: N
- Caso médio: espera-se $2lgN$
- `https://fga.rysh.com.br/eda1/pdf/5-tad.pdf`

Árvore binária de busca - Performance

- Árvores não balanceadas espera-se caso médio $2lgN$
  - Sendo o primeiro elemento a raiz
  - Chaves aleatoriamente inseridas
  - Totalmente balanceadas, raras
  - Totalmente desbalanceadas, raras
    - ★ Exemplo de inserção: 1 2 3 4 5 6 7