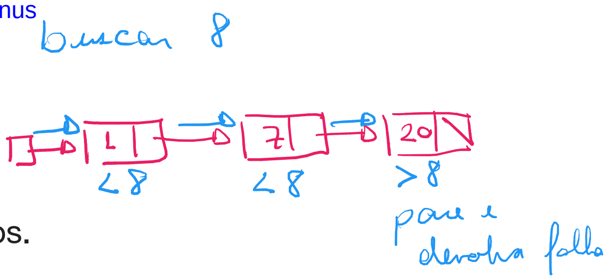
05 - Bônus

AED2 - Aula ~~06~~

Skip lists



Considere usar uma lista ligada ordenada

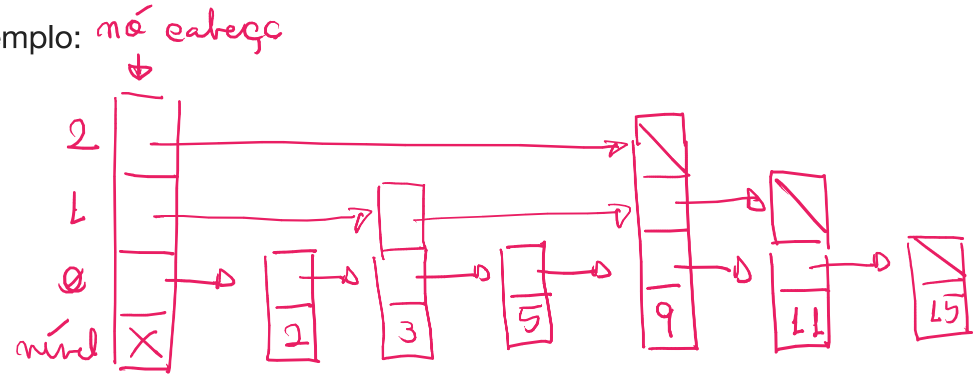
- para implementar uma tabela de símbolos.

- Numa busca percorremos a lista até encontrar
  - uma chave maior ou igual que a buscada
    - ou até chegar no fim da lista.
- Essa abordagem não é eficiente, pois localizar um registro,
  - no pior caso, leva tempo proporcional ao tamanho da lista, i.e., O(n)

A ideia da estrutura de dados Skip List é usar

- uma hierarquia de listas ligadas ordenadas conectadas entre si,
  - em que cada lista tem uma quantidade de itens diferente,
    - de acordo com o fator de dispersão da estrutura.

Exemplo:

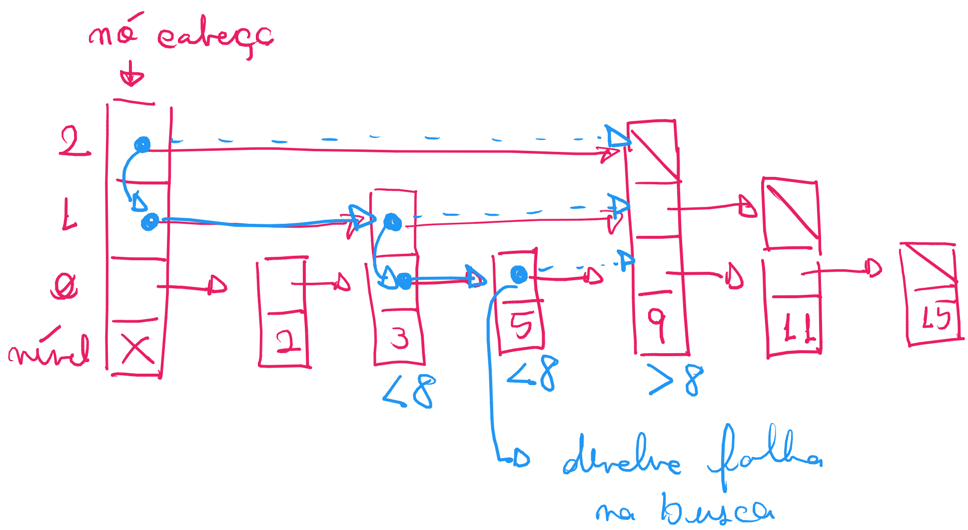


- Note que um item de nível alto sempre aparece nos níveis mais baixos.
- Importante salientar que essa é uma estrutura aleatorizada/probabilística.
  - O motivo para isso veremos mais à frente.

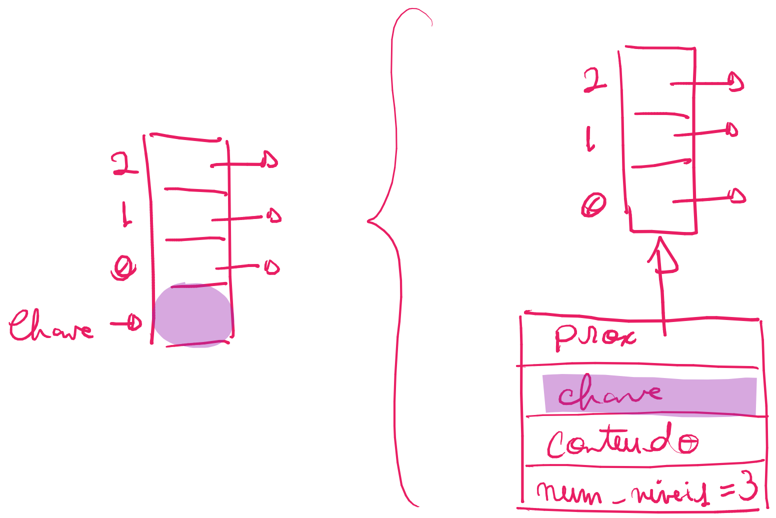
Ideia da busca: começar a percorrer a lista do maior nível, que tem menos itens.

- Quando encontrar o fim da lista
  - ou um item com chave maior do que a buscada,
- continua a busca no nível abaixo, que tem mais itens.

Exemplo de busca pela chave 8:



Antes de analisarmos o código da busca, vamos entender a estrutura dos nós.




```
#define num_max_niveis 100

typedef int Chave;
typedef int Item;

typedef struct noh {
   Chave chave;
   Item conteudo;
   struct noh **prox;
   int num_niveis;
} Noh;

static Noh *lista;
static int num_itens, nivel_max;
```

→ vetor de listas

} variáveis globais p/ simplificar

Código da busca:

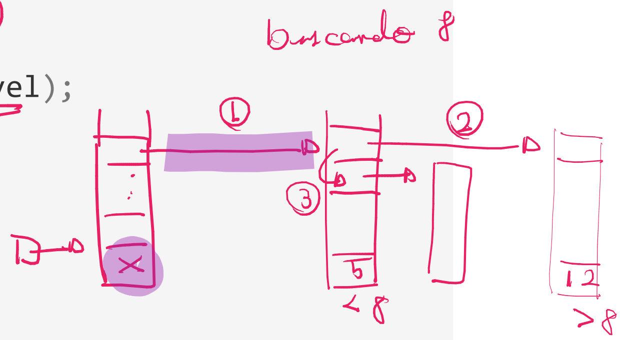
```
Noh *buscaR(Noh *t, Chave chave, int nivel) {
   if (t != lista && chave == t->chave) return t;
   if (t->prox[nivel] == NULL || chave < t->prox[nivel]->chave) {
      if (nivel == 0) return NULL;
      return buscaR(t, chave, nivel - 1);
   }
   return buscaR(t->prox[nivel], chave, nivel);
}

Noh *TSbusca(Chave chave) {
   return buscaR(lista, chave, nivel_max);
}
```

verificando se não é o nó cabeça

→ encontrou

→ não encontrou nesse nível

→ busca falhou



Quiz: Transformar essa busca recursiva em iterativa.

Definimos o fator de dispersão como um inteiro $d \geq 2$, que indica

- que o número de nós do nível i para o nível $i+1$ cai, em média, de $1/d$

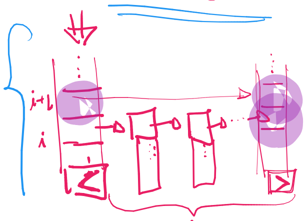
Eficiência de tempo: observe que uma skip list com n itens deve ter $1 + \log_d n$ níveis,

  - já que o número de itens cai de $1/d$ por nível.
- Além disso, entre dois nós do nível $i+1$ devem existir,
  - em média, d valores no nível i
- Por isso, esperamos dar $d/2$ saltos por nível, em média,
  - antes de descer para o nível seguinte.
- Assim, o tempo esperado de busca é $(d/2)\log_d n = O(\log n)$
  - derivado do produto do número esperado de níveis
    - pelo número esperado de saltos por nível.
- Por ser uma estrutura probabilística, falamos em eficiência esperada.
  - Mas essa média depende apenas das escolhas aleatórias
    - da própria estrutura, e não dos valores da entrada.

$n/d^k = 1$
$\vdots$
n/d^2
n/d
n

$d^k = n$

$K = \log_d n$



d nós

Eficiência de espaço: skip lists tem, em média, $n \cdot d/(d-1) = O(n)$ blocos. de memória

- Para entender de onde vem esse valor, observe que,
  - o primeiro nível tem n nós, o segundo tem n/d, o terceiro n/d^2, ...
- Assim, o número esperado de ~~nós~~ blocos de memória corresponde
  - à soma dos termos de uma Progressão Geométrica (PG)
    - que começa em n e tem razão $1/d$
- Deduzindo a soma dos termos de uma PG de razão $q < 1$ temos

$\text{Soma PG}(q) = 1 + q + q^2 + q^3 + \dots$

$(-)$

$q \, \text{Soma PG}(q) = q + q^2 + q^3 + \dots$

$(1-q)\,\text{Soma PG}(q) = (1 + \cancel{q} + \cancel{q^2} + \dots) - (\cancel{q} + \cancel{q^2} + \dots) = 1$

$\text{Soma PG}(q) = 1/(1-q)$

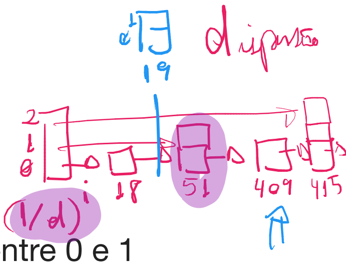
- Como na nossa PG o primeiro termo é n e a razão é $1/d$
  - sua soma converge para $n \cdot 1/(1 - 1/d) = \frac{n}{\frac{d-1}{d}} = n \cdot \frac{d}{(d-1)} = O(n)$

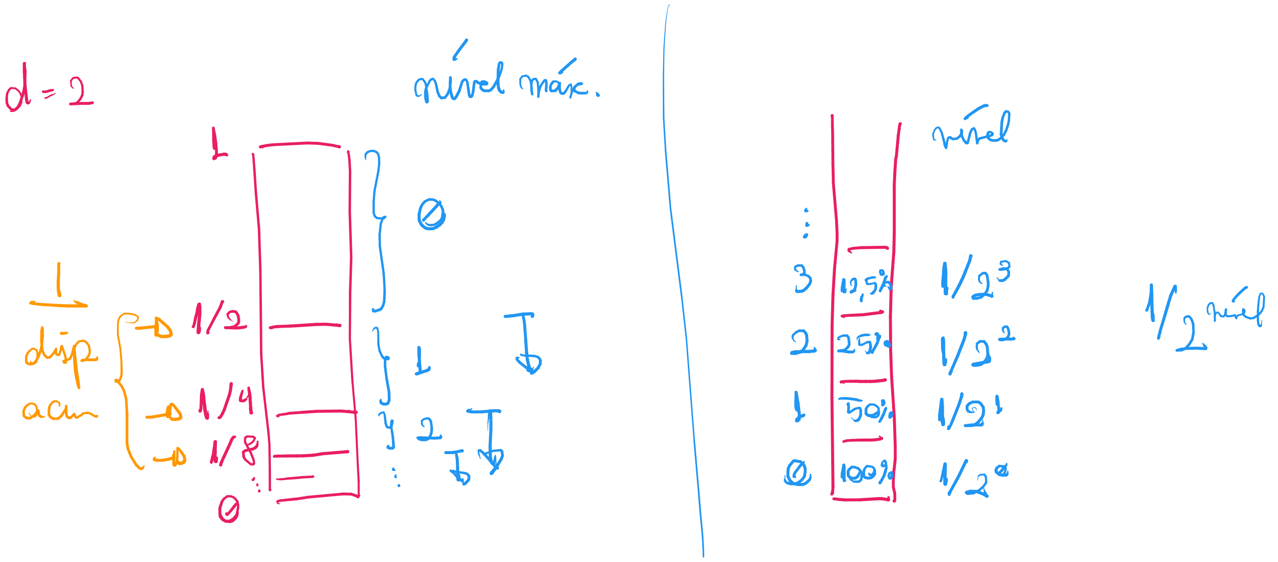
Relação entre tempo e espaço: dado um fator de dispersão d constante, temos

- eficiência de tempo $d(\log_d n)/2 = O(\log n)$
  - e eficiência de espaço $n \cdot d/(d-1) = O(n)$
- Agora, vamos comparar skip lists com diferentes fatores de dispersão.
  - Como exemplo, tome $d' = 2$
    - tempo = $2 \cdot (\lg n)/2 = \lg n$
    - espaço = $n \cdot 2/(2-1) = 2n$
  - e compare com $d'' = 10$
    - tempo = $10(\log_{10} n)/2 = \frac{10}{2}\left(\frac{\log_2 n}{\log_2 10}\right) = \frac{5}{\log_2 10} \cdot \lg n \approx 1{,}5 \lg n$
    - espaço = $n \cdot 10/(10-1) \approx 1{,}11 n$
- Podemos perceber que, quanto maior o fator de dispersão d
  - mais lenta é a busca e menos espaço ocupa a skip list.

Probabilidade: a ideia central é que a cada novo nível temos menos nós,

- mais especificamente $1/d$ do número de nós do nível anterior,
  - e estes devem estar homogeneamente espaçados.
- Para obter tal resultado precisamos utilizar escolhas aleatórias,
  - de modo que um nó pertença ao nível i com probabilidade $(1/d)^i$
- Para fazer isso, sem jogar múltiplas moedas, sorteamos um valor entre 0 e 1
  - e atribuímos um nível de acordo com o valor sorteado.





- A seguinte função implementa essa ideia

```
int nivelAleatorio() {
   int nivel, disp_acum, d = 2, v = rand();
   disp_acum = d;
   for (nivel = 0; nivel < num_max_niveis; nivel++) {
      if (v > RAND_MAX / disp_acum) break;
      disp_acum *= d;
   }
   if (nivel > nivel_max) nivel_max = nivel;
   return nivel;
}
```

$\in [0, RAND\_MAX]$

$= disp\_acum * d$

$\frac{v}{RAND\_MAX} > \frac{1}{disp\_acum}$

- Observe que, essa função sorteia um valor v e
  - quanto menor tal valor maior será o nível do nó.
- Note que, no início do laço valem os invariantes

$disp\_acum = d^{(nivel+1)}$

$\frac{v}{RAND\_MAX} \leq \frac{1}{d^{nivel}}$

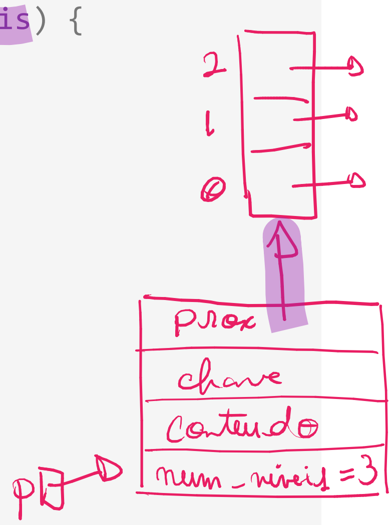
- Como $v = rand()$ recebe um valor aleatório e uniforme entre 0 e $RAND\_MAX$
  - o que estamos fazendo é sortear um valor entre 0 e 1
- e colocando o nó num nível j se o valor sorteado for $\leq 1/d^j$
  - o que ocorre com probabilidade $1/d^j$

Ideia da inserção: sortear um nível para o novo nó,

- percorrer um caminho semelhante ao da busca até
  - chegar na posição que o nó deveria ocupar no nível sorteado,
- então inserir o nó em todas as listas com nível menor ou igual ao dele.

Código da inserção:

```
void TSinsere(Chave chave, Item conteudo) {
    int nivelAleat = nivelAleatorio();
    Noh *novoNoh = novo(chave, conteudo, nivelAleat + 1);
    insereR(lista, novoNoh, nivel_max);
    num_itens++;
}

Noh *novo(Chave chave, Item conteudo, int num_niveis) {
    int i;
    Noh *p = (Noh *)malloc(sizeof *p);
    p->chave = chave;
    p->conteudo = conteudo;
    p->prox = malloc(num_niveis * sizeof(Noh *));
    p->num_niveis = num_niveis;
    for (i = 0; i < num_niveis; i++)
        p->prox[i] = NULL;
    return p;
}
```

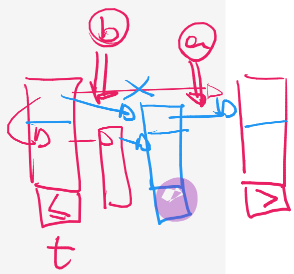



```
void insereR(Noh *t, Noh *novoNoh, int nivel) {
    Chave chave = novoNoh->chave;
    if (t->prox[nivel] == NULL || chave < t->prox[nivel]->chave) {
        if (nivel < novoNoh->num_niveis) {
            novoNoh->prox[nivel] = t->prox[nivel];    (a)
            t->prox[nivel] = novoNoh;                 (b)
        }
        if (nivel > 0) insereR(t, novoNoh, nivel - 1);
        return;
    }
    insereR(t->prox[nivel], novoNoh, nivel);
}
```

não contém nesse nível

devo inserir o novo nó nesse nível?

continuo a "busca" no mesmo nível



Eficiência de tempo: inserção faz, em média, $(d/2)\log_d n = O(\log n)$ comparações,

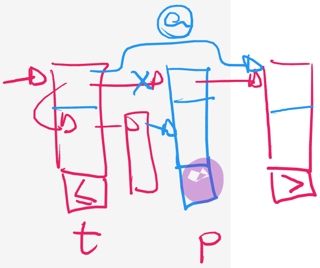
- sendo $d \geq 2$ o fator de dispersão da skip list.

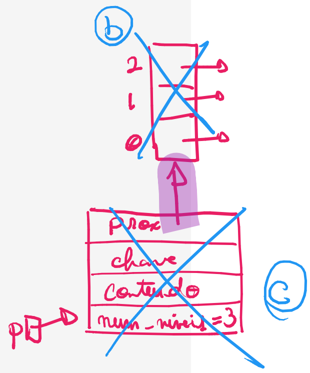
Ideia da remoção: a ideia é buscar o nó normalmente,

- ○ removê-lo de todas as listas com nível menor ou igual ao dele,
- e então liberar o nó em si.

Código da remoção:

```
int removeR(Noh *t, Chave chave, int nivel) {
   Noh *p = t->prox[nivel];
   if (p == NULL || chave <= p->chave) {     não devo continuar a busca nesse nível
      if (p != NULL && chave == p->chave) {
         t->prox[nivel] = p->prox[nivel];
         if (nivel == 0) {
            free(p->prox);
            free(p);
            return 1;  "sucesso"
         }
      }
      if (nivel == 0) return 0;  "fracasso"
      return removeR(t, chave, nivel - 1);
   }
   return removeR(t->prox[nivel], chave, nivel);
}

void TSremove(Chave chave) {
   if (removeR(lista, chave, nivel_max))
      num_itens--;
}
```





\# de passos por nível   \# de níveis

Eficiência de tempo: remoção faz, em média, $(d/2)\log_d n = O(\log n)$ comparações,

- sendo $d \geq 2$ o fator de dispersão da skip list.

Bônus: observe que, nas várias funções as chamadas recursivas

- ○ são feitas por último, o que caracteriza recursão caudal.
- Neste caso, é relativamente simples substituir essas chamadas recursivas
  - ○ por um laço, que envolve a função e tem condições de parada
    - ■ complementares ao caso base da recursão,
  - ○ junto da reatribuição de valores para os parâmetros da função
    - ■ nos pontos em que ocorriam as chamadas recursivas.
- Desafio: implementar a versão iterativa das funções da skip list.