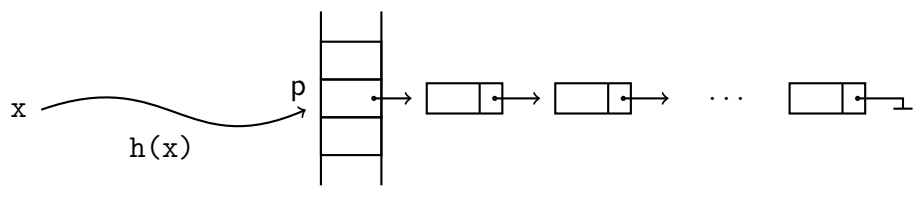
## Análise

É fácil ver que o procedimento de *inserção* executa em tempo $O(1)$
— (*ele se comporta como a inserção em uma lista desordenada*)

A *busca* e a *remoção* na tabela de dispersão também se comportam como na lista desordenada.

Portanto, o seu tempo de execução é proporcional ao tamanho da lista associada à posição `p` da tabela




Considerando o cenário de pior caso, a função de dispersão `h`(.) mapeia todos os elementos da tabela para a posição `p`

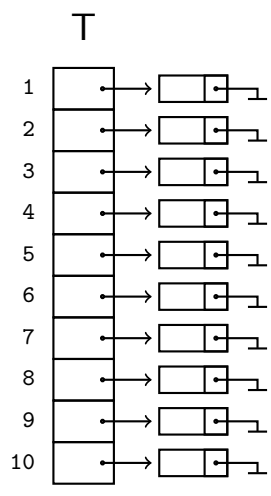
E nesse caso, tanto a busca como a remoção levam tempo $O(n)$.

Mas dessa vez, a suposição de pior caso não é razoável.

Quer dizer,

→ *Nenhuma função de dispersão digna desse nome mapearia todos os elementos para a mesma posição da tabela*
— *(uma função assim deveria se chamar função de concentração ...)*

Outra possibilidade é considerar a função de dispersão perfeita, que mapeia todos os elementos para posições diferentes.

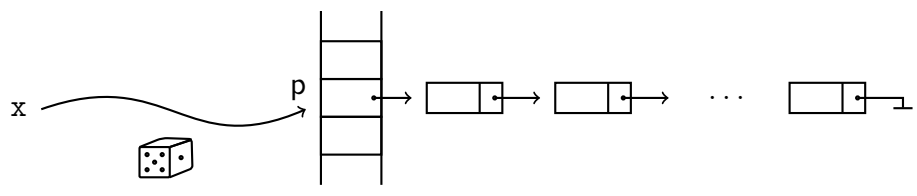



Mas essa suposição tampouco é razoável, na prática.

Na prática, o mais razoável é imaginar que uma boa função de dispersão se comporta como uma *função aleatória*.

*Mas o que é isso?*

Bom, a ideia é que a função aleatória lança um dado com $n$ faces para determinar a posição de cada elemento na tabela




— (*i.e., todas as posições tem a mesma probabilidade de serem obtidas no lançamento do dado*)

*Sim, mas o que significa isso?*

Bom, isso significa que ainda vão haver colisões.

E por causa disso, vão haver posições da tabela com mais de um elemento.

E também vão haver posições da tabela sem elemento nenhum — (*assumindo que nós temos uma tabela de tamanho n que armazena n elementos*)

Algumas posições da tabela vão conter apenas 1 elemento.

Outras posições vão conter 2 elementos.

E por aí vai ...

Certo.

*Mas, e daí?*

Bom, daí que assumindo que a função de dispersão tem comportamento aleatório, nós podemos fazer várias observações interessantes — (*que hoje vão ficar sem prova ...*)

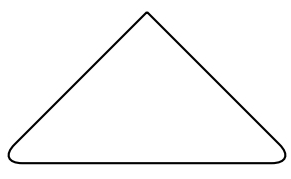
1. 63,2% dos elementos estão na primeira posição das suas listas,
   e são encontrados imediatamente pela busca — (*i.e., 1 comparação*)
2. quase 90% dos elementos estão na primeira ou segunda posição das suas listas,
   e são encontrados com até 2 comparações
3. quase 98% dos elementos estão na primeira, segunda ou terceira posição das suas listas,
   e são encontrados com até 3 comparações
4. 99,9% dos elementos estão na primeira, segunda, terceira ou quarta posição das suas listas,
   e são encontrados com até 4 comparações
5. a maior lista de todas contém (aproximadamente) $\ln n$ elementos
   daí que, no pior caso, a busca leva tempo $O(\ln n)$
   — (*assumindo que a função de dispersão tem comportamento aleatório*)

Quer dizer, a busca quase sempre executa em tempo $O(1)$.

E nas poucas vezes em que ela apresenta o pior caso, a busca leva tempo $O(\ln n)$.

Aqui vale a pena fazer o contraste com a árvore binária de busca.

Quer dizer, mesmo na situação ideal em que nós temos uma árvore completa



- a busca quase sempre executa em tempo $O(\log n)$
  — (*porque a vasta maioria dos elementos está na parte de baixo da árvore*)
- e nas poucas vezes em que o elemento é encontrado próximo da raiz, a busca executa em tempo $O(1)$