Estruturas de dados

Programação Avançada
Universidade Federal de Pernambuco
Adrien Durand-Petiteville
adrien.durandpetiteville@ufpe.br

Introdução

- Conjuntos são tão fundamentais para a Ciência da Computação quanto para a Matemática.
- Enquanto os conjuntos matemáticos são invariáveis, os conjuntos manipulados por algoritmos podem crescer, encolher ou sofrer outras mudanças ao longo do tempo.
- Chamamos tais conjuntos de conjuntos dinâmicos.
- Nessa parte, apresentamos algumas técnicas básicas para representar conjuntos dinâmicos finitos e para manipular esses conjuntos em um computador.
- Algoritmos podem exigir a execução de vários tipos diferentes de operações em conjuntos.
- Por exemplo, muitos algoritmos precisam apenas da capacidade de inserir e eliminar elementos em um conjunto e testar a pertinência de elementos a um conjunto.
- A melhor maneira de implementar um conjunto dinâmico depende das operações que devem ser suportadas.

Elementos de um conjunto dinâmico

- Em uma implementação típica de um conjunto dinâmico, cada elemento é representado por um objeto cujos atributos podem ser examinados e manipulados se tivermos um ponteiro para o objeto.
- Alguns tipos de conjuntos dinâmicos consideram que um dos atributos do objeto é uma chave de identificação.
- O objeto pode conter dados satélites, que são transportados em atributos de outro objeto mas que, fora isso, não são utilizados pela implementação do conjunto.
- Também pode ter atributos que são manipulados pelas operações de conjuntos; esses atributos podem conter dados ou ponteiros para outros objetos no conjunto.
- Alguns conjuntos dinâmicos pressupõem que as chaves são extraídas de um conjunto totalmente ordenado como o dos números reais ou o de todas as palavras sob a ordenação alfabética usual.

Operações em conjuntos dinâmicos

- As operações em um conjunto dinâmico podem ser agrupadas em duas categorias:
  - Consultas, que simplesmente retornam informações sobre o conjunto
  - Operações modificadoras, que alteram o conjunto
- Apresentamos a seguir, uma lista de operações típicas. Qualquer aplicação específica, normalmente exigirá a implementação de apenas algumas dessas operações.

Operações em conjuntos dinâmicos - 2

- `SEARCH(S,k)`
  Uma consulta que, dado um conjunto `S` e um valor de chave `k`, retorna um ponteiro `x` para um elemento em `S` tal que `x.chave = k` ou `NIL` se nenhum elemento desse tipo pertencer a `S`.
- `INSERT (S,x)`
  Uma operação modificadora que aumenta o conjunto `S` com o elemento apontado por `x`.
  Normalmente, consideramos que quaisquer atributos no elemento `x` necessários para a implementação do conjunto já foram inicializados.
- `DELETE(S,x)`
  Uma operação modificadora que, dado um ponteiro `x` para um elemento no conjunto `S`, remove `x` de `S`.
  (Observe que essa operação utiliza um ponteiro para um elemento `x`, não um valor de chave.)

Operações em conjuntos dinâmicos - 3

- `MININUM(S)`
  Uma consulta em um conjunto totalmente ordenado `S` que retorna um ponteiro para o elemento de `S` que tenha a menor chave.
- `MAXIMUM(S)`
  Uma consulta em um conjunto totalmente ordenado `S` que retorna um ponteiro para o elemento de `S` que tenha a maior chave.
- `SUCCESSOR (S,x)`
  Uma consulta que, dado um elemento `x` cuja chave é de um conjunto totalmente ordenado `S`, retorna um ponteiro para o elemento maior seguinte em `S` ou `NIL` se `x` é o elemento máximo.
- `PREDECESSOR(S,x)`
  Uma consulta que, dado um elemento `x`, cuja chave é de um conjunto totalmente ordenado `S`, retorna um ponteiro para o elemento menor seguinte em `S` ou `NIL` se `x` é o elemento mínimo.

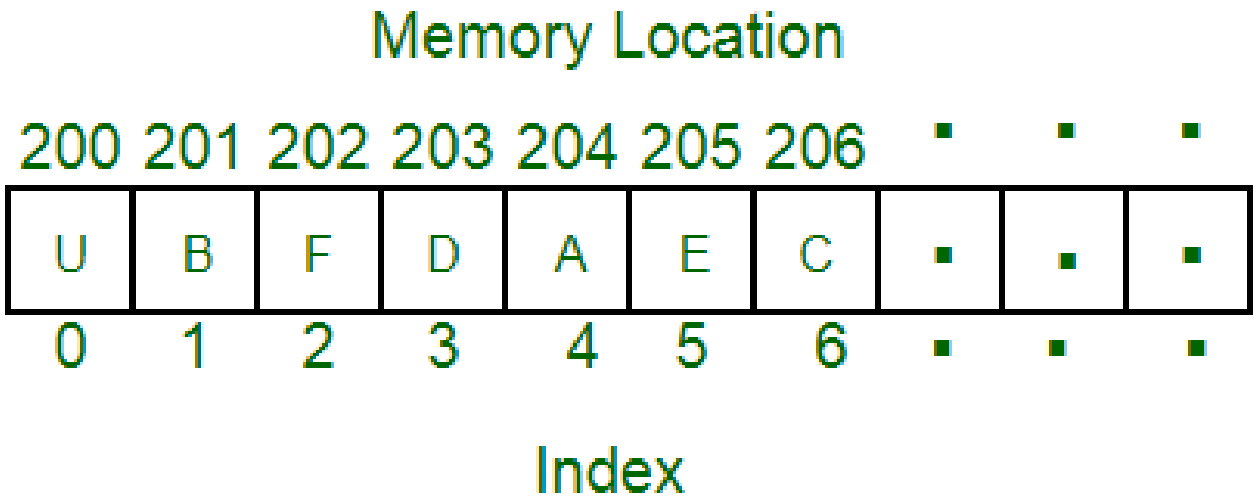
Estruturas de dados

- Uma estrutura de dados é uma forma particular de organizar os conjuntos em um computador para que possam ser usados eficazmente.
- A ideia é reduzir as complexidades de espaço e tempo de diferentes tarefas.
- Abaixo está uma lista de algumas estruturas de dados lineares populares.
  - Arranjo
  - Lista ligada
  - Pilha
  - Fila
- Uma estrutura de dados linear é uma estrutura na qual os objetos estão organizados em ordem linear.

Arranjos

Arranjo

- Um arranjo é uma coleção de itens armazenados em locais de memória contígua.
- A ideia é armazenar vários itens do mesmo tipo juntos.
- Isto facilita o cálculo da posição de cada elemento simplesmente adicionando um offset a um valor base, ou seja, a localização da memória do primeiro elemento do arranjo (geralmente denotado pelo nome do arranjo).



Vantagens e desvantagens do uso de arranjos

- Vantagens do uso de arranjos:
  - Os arranjos permitem o acesso aleatório aos elementos. Isto torna o acesso aos elementos por posição mais rápido.
  - Os arranjos têm melhor localização de cache que pode fazer uma grande diferença no desempenho.
  - Os arranjos representam vários itens de dados do mesmo tipo usando um único nome.
- Desvantagens do uso de arranjos:
  - Você não pode mudar o tamanho, ou seja, uma vez declarado o arranjo, você não pode mudar seu tamanho por causa da memória estática alocada a ele.
  - Aqui a inserção e exclusão são difíceis, pois os elementos são armazenados em locais de memória consecutivos e a operação de deslocamento também é dispendiosa.
- Aplicações de arranjos
  - Arranjo armazena elementos de dados do mesmo tipo de dados.
  - Usados para implementar outras estruturas de dados como pilhas, filas, pilhas, tabelas de hastes, etc.

Listas ligadas

Listas ligadas

- Uma lista ligada é uma estrutura de dados linear (como arranjos) onde cada elemento é um objeto separado.
- Cada elemento (que é um nó) de uma lista é composto de dois itens: os dados e uma referência ao próximo nó.
- Entretanto, diferentemente de um arranjo, no qual a ordem linear é determinada pelos índices do arranjo, a ordem em uma lista ligada é determinada por um ponteiro em cada objeto.
- Listas ligadas nos dão uma representação simples e flexível para conjuntos dinâmicos, suportando (embora não necessariamente com eficiência) todas as operações que aparecem na lista da introdução.

Tipos de listas

- Uma lista pode ter uma entre várias formas.
- Ela pode ser **simplesmente** ligada ou **duplamente** ligada, pode ser **ordenada** ou não e pode ser **circular** ou não.
  - Se uma lista é **simplesmente** ligada, omitimos o ponteiro anterior em cada elemento.
    Simplesmente ligada: `1->2->3->4->NULL`
    Duplamente ligada: `NULL<-1<->2<->3->NULL`
  - Em uma lista **circular**, o ponteiro anterior do início da lista aponta para o fim, e o ponteiro próximo do fim da lista aponta para o início.
    Circular: `1->2->3->1`
  - Podemos imaginar uma lista circular como um anel de elementos.

Tipos de listas - 2

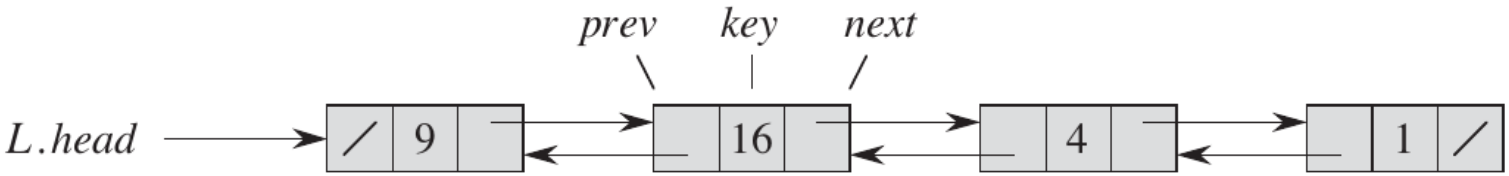
- Se uma lista é **ordenada**, a ordem linear da lista corresponde à ordem linear de chaves armazenadas em elementos da lista; então, o elemento mínimo é o início da lista, e o elemento máximo é o fim.
- Se a lista é não ordenada, os elementos podem aparecer em qualquer ordem.
- **No restante desta seção, supomos que as listas com as quais estamos trabalhando são listas não ordenadas e duplamente ligadas.**

Lista duplamente ligada

- Cada elemento de uma lista duplamente ligada `L` é um objeto com um atributo chave e dois outros atributos ponteiros: próximo e anterior.
- O objeto também pode conter outros dados satélites.
- Dado um elemento `x` na lista, `x.next` aponta para seu sucessor na lista ligada e `x.prev` aponta para seu predecessor.
- Se `x.prev = NIL`, o elemento `x` não tem nenhum predecessor e, portanto, é o primeiro elemento, ou início, da lista.
- Se `x.next = NIL`, o elemento `x` não tem nenhum sucessor e, assim, é o último elemento, ou fim, da lista.
- Um atributo `L.head` aponta para o primeiro elemento da lista.
- Se `L.head = NIL`, a lista está vazia.

Exemplo

- Uma lista duplamente ligada `L` representando o conjunto dinâmico 1, 4, 9, 16.
- Cada elemento na lista é um objeto com atributos para a chave e ponteiros (mostrados por setas) para o próximo objeto e para o objeto anterior.
- O atributo próximo do fim e o atributo anterior do início são `NIL`, indicados por uma barra diagonal.
- O atributo `L.head` aponta para o início.



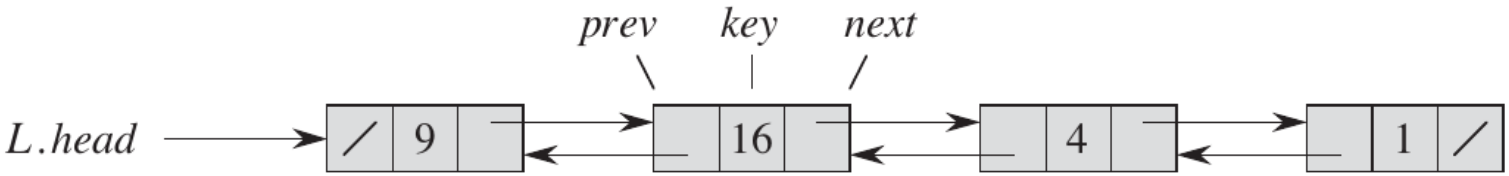
Fazer uma busca em uma lista ligada

- O procedimento `LIST-SEARCH(L,k)` encontra o primeiro elemento com chave `k` na lista `L` por meio de uma busca linear simples, retornando um ponteiro para esse elemento.
- Se nenhum objeto com chave `k` aparecer na lista, o procedimento retorna `NIL`.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

Exemplo

- No caso da lista ligada da figura:
  - A chamada `LIST-SEARCH(L,4)` retorna um ponteiro para o terceiro elemento.
  - A chamada `LIST-SEARCH (L,7)` retorna `NIL`.



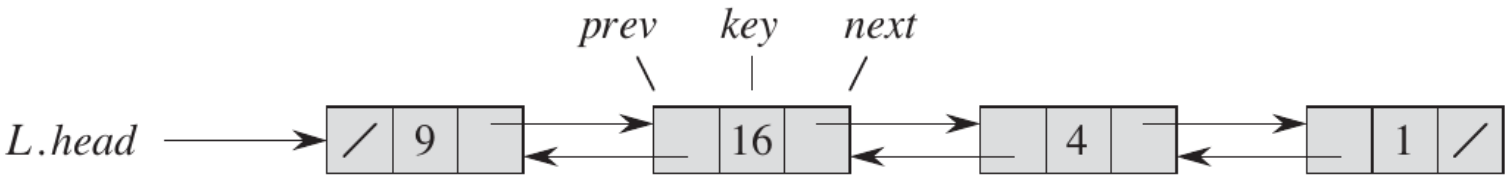
Inserção em uma lista ligada

- Dado um elemento `x` cujo atributo chave já foi definido, o procedimento `LIST-INSERT` "emenda" `x` à frente da lista ligada

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

Exemplo

- Lista inicial



- Seguindo a execução de `LIST-INSERT(L,x)`, onde `x.chave = 25`, a lista ligada tem um novo objeto com chave 25 como o novo início.
- Esse novo objeto aponta para o antigo início com chave 9.

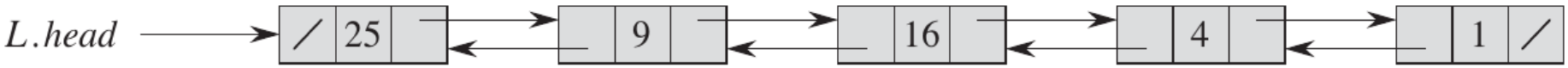
Eliminação em uma lista ligada

- O procedimento `LIST-DELETE` remove um elemento `x` de uma lista ligada `L`.
- Ele deve receber um ponteiro para `x`, e depois "desligar" `x` da lista atualizando os ponteiros.
- Se desejarmos eliminar um elemento com determinada chave, deveremos primeiro chamar `LIST-SEARCH`, para reaver um ponteiro para o elemento.

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

Exemplo

- Lista inicial



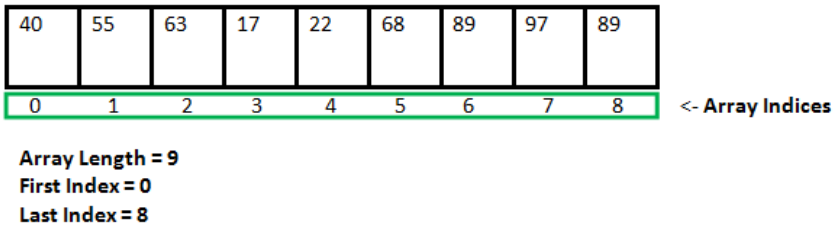
- O resultado da chamada subsequente `LIST-DELETE(L,x)`, onde `x` aponta para o objeto com chave 4

Vantagens e desvantagens do uso de listas ligadas

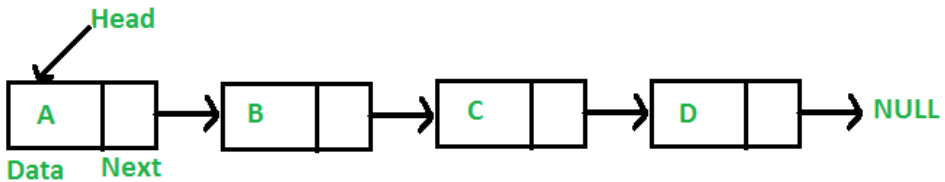
- Vantagens do uso de listas ligadas:
  - Tamanho dinâmico.
  - Facilidade de inserção/deleção.
- Desvantagens do uso de listas ligadas:
  - O acesso aleatório não é permitido. Temos que acessar os elementos sequencialmente a partir do primeiro nó.
  - É necessário espaço extra de memória para um ponteiro com cada elemento da lista.
  - Não é amigável ao cache. Como os elementos do arranjo são locais contíguos, há uma localidade de referência que não existe no caso de listas vinculadas.

Arranjos vs Lista ligadas

- As arranjos armazenam elementos em locais de memória contíguos, resultando em endereços facilmente calculáveis para os elementos armazenados e isto permite um acesso mais rápido a um elemento em um índice específico.



- As listas ligadas são menos rígidas em sua estrutura de armazenamento e os elementos geralmente não são armazenados em locais contíguos, portanto, elas precisam ser armazenados com etiquetas adicionais dando uma referência ao próximo elemento.



- Esta diferença no esquema de armazenamento de dados decide qual estrutura de dados seria mais adequada para uma determinada

Implementação

```
class Node {
  public:
  int data;
  Node* next;
};
```

Exemplo

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;
};

// This function prints contents of linked list starting from the give
void printList(Node* n)
{
    while (n != NULL) {
        cout << n->data << endl;
        n = n->next;
    }
}
```

Exemplo - 2

```
int main()
{
    // Head of the list - initially pointing NUll
    Node* head = NULL;

    // Allocate 3 nodes
    Node* first = new Node();
    Node* second = new Node();
    Node* third = new Node();


    // Insert first node
    first->data = 1; // assign data
    first->next = head; // point to next node
    head = first; // update head
```

Exemplo - 3

```
    // Insert second node
    second->data = 2; // assign data
    second->next = head; // point to next node
    head = second; // update head

    // Insert third node
    third->data = 3; // assign data
    third->next = head; // point to next node
    head = third; // update head

    printList(head);

    return 0;
}
```

Pilhas

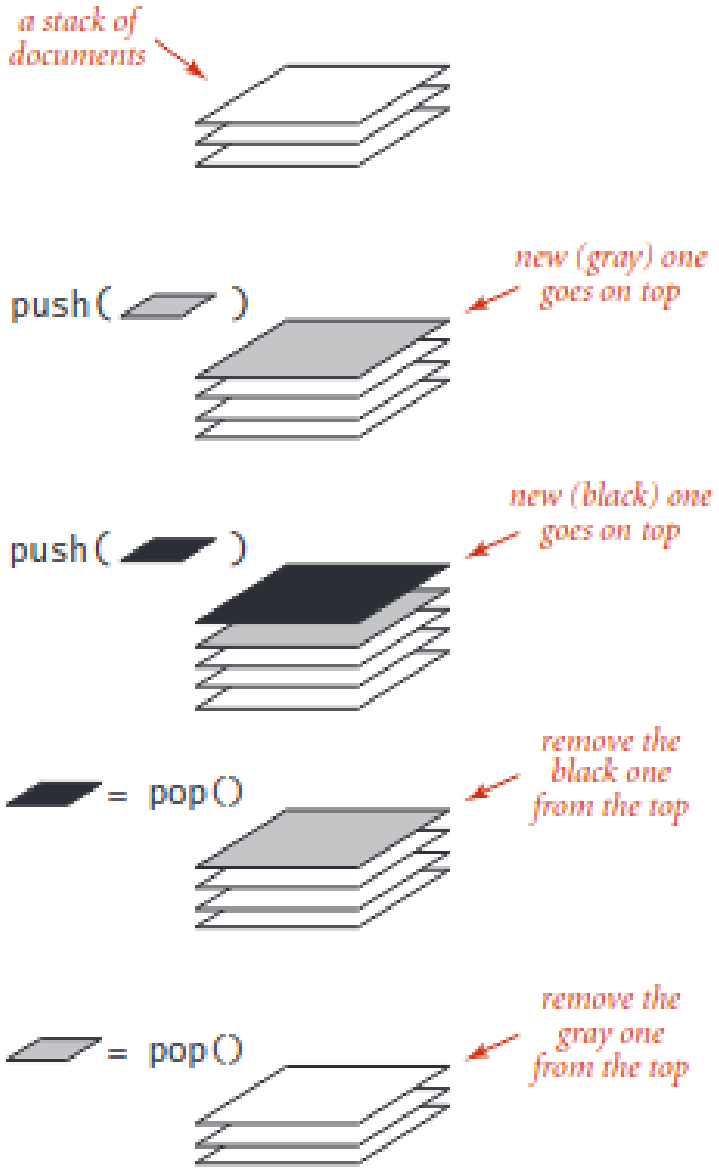
Pilhas

- Pilhas são conjuntos dinâmicos nos quais o elemento removido do conjunto pela operação `DELETE` é especificado previamente.
- Em uma pilha, o elemento eliminado do conjunto é o mais recentemente inserido
- A pilha implementa uma política de **último a entrar, primeiro a sair** ou **LIFO** (last-in, first-out).

Pilhas - 2

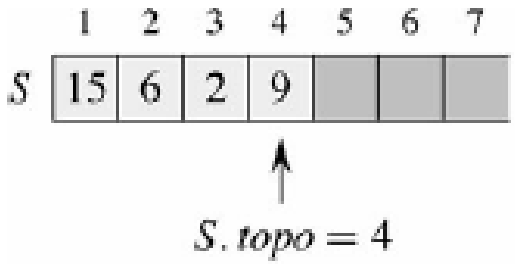
- A operação `INSERT` em uma pilha é frequentemente denominada `PUSH`.
- A operação `DELETE`, que não toma um argumento de elemento, é frequentemente denominada `POP`.
- Esses nomes são alusões a pilhas físicas, como as pilhas de pratos acionadas por molas usadas em restaurantes:
  A ordem em que os pratos são retirados da pilha é o inverso da ordem em que foram colocados na pilha, já que apenas o prato do topo está acessível.

Pilhas - 3


a stack of documents
push( )
new (gray) one goes on top
push( )
new (black) one goes on top
= pop()
remove the black one from the top
= pop()
remove the gray one from the top

Atributos de uma pilha

- Podemos implementar uma pilha de no máximo `n` elementos com um arranjo `S[1.. n]`.
- O arranjo tem um atributo `S.topo` que indexa o elemento mais recentemente inserido.
- A pilha consiste nos elementos `S[1 ... S.topo]`, onde `S[1]` é o elemento na parte inferior da pilha e `S[S.topo]` é o elemento na parte superior.



Atributos de uma pilha - 2

- Quando `S.topo = 0`, a pilha não contém nenhum elemento e está vazia.
- Podemos testar se a pilha está vazia pela operação de consulta `STACKEMPTY`.
- Se tentarmos extrair algo de uma pilha vazia, dizemos que a pilha tem estouro negativo, que é normalmente um erro.
- Se `S.topo` exceder `n`, a pilha tem um estouro. (Em nossa implementação de pseudocódigo, não nos preocuparemos com estouro de pilha.)

Pseudocódigo

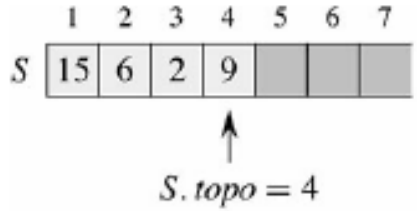
```
STACK-EMPTY(S)
1  if S.topo == 0
2      return TRUE
3  else return FALSE
```

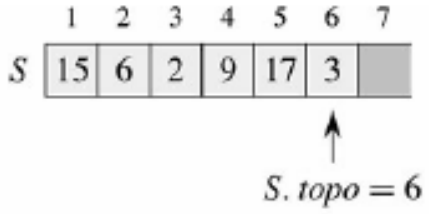
```
PUSH(S, x)
1  S.topo = S.topo + 1
2  S[S.topo] = x
```

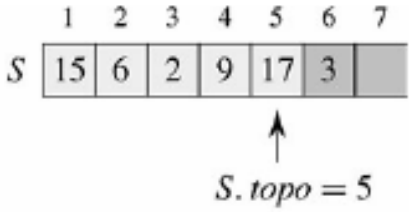
```
POP(S)
1  if STACK-EMPTY(S)
2      error "underflow"
3  else S.topo = S.topo − 1
4      return S[S.topo + 1]
```

Exemplo

- (a)
  - A pilha `S` tem quatro elementos.
  - O elemento do topo é 9.
- (b)
  - A pilha `S` após as chamadas `PUSH(S,17)` e `PUSH(S,3)`.
- (c)
  - A pilha `S` após a chamada `POP(S)` retornou o elemento 3, que é o elemento mais recentemente inserido na pilha.
  - Embora ainda apareça no arranjo, o elemento 3 não está mais na pilha
  - O topo é o elemento 17.







Aplicações de pilhas:

- A Redo-undo apresenta-se em muitos lugares como editores, photoshop.
- Recursos para frente e para trás em navegadores web
- No gerenciamento de memória, qualquer computador moderno usa a pilha como a gestão primária para uma finalidade de execução. Cada programa que está sendo executado em um sistema de computador tem suas próprias alocações de memória
- A inversão de `string` é também uma outra aplicação da pilha. Aqui um a um cada caractere é inserido na pilha. Assim, o primeiro caractere do `string` está na parte inferior da pilha e o último elemento do `string` está na parte superior da pilha. Depois de realizar as operações de pop na pilha, obtemos o `string` em ordem inversa.

Implementação de pilhas

- Implementação de pilhas usando arranjo
  - **Prós**: Fácil de implementar. A memória é salva porque os ponteiros não estão envolvidos.
  - **Cons**: Não é dinâmico. Não cresce e encolhe dependendo das necessidades em tempo de execução.
- Implementação de pilhas usando lista
  - **Prós**: A pilha pode crescer e encolher de acordo com as necessidades em tempo de execução.
  - **Cons**: Requer memória extra devido ao envolvimento de apontadores.

Filas

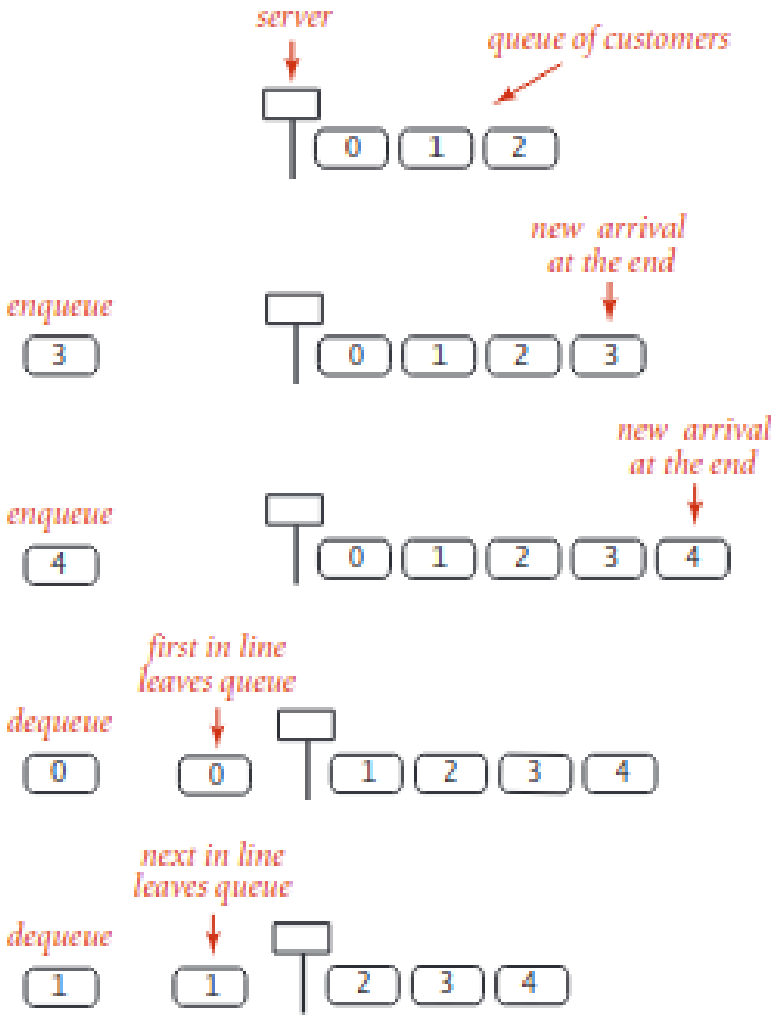
Filas

- Filas são conjuntos dinâmicos nos quais o elemento removido do conjunto pela operação `DELETE` é especificado previamente.
- Em uma fila o elemento eliminado é sempre o que estava no conjunto há mais tempo.
- A fila implementa uma política de **primeiro a entrar, primeiro a sair** ou **FIFO** (first-in, first-out)

Filas - 2

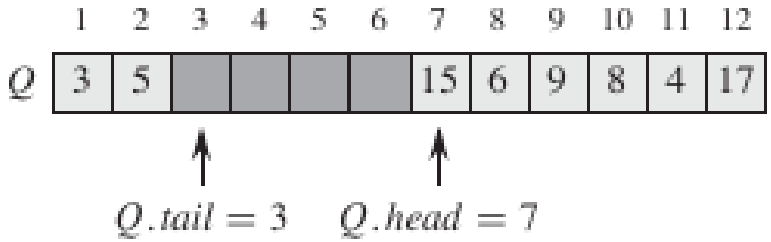
- Designamos a operação `INSERT` em uma fila por `ENQUEUE` (`ENFILEIRAR`) e a operação `DELETE` por `DEQUEUE` (`DESINFILEIRAR`)
- Assim como a operação em pilhas `POP`, `DEQUEUE` não adota nenhum argumento de elemento.
- A propriedade `FIFO` de uma fila faz com que ela funcione como uma fileira de pessoas em uma caixa registradora.
- A fila tem um **início** (ou cabeça) e um **fim** (ou cauda).
- Quando um elemento é inserido na fila, ocupa seu lugar no fim da fila, exatamente como um cliente que acabou de chegar ocupa um lugar no final da fileira.
- O elemento retirado da fila é sempre aquele que está no início da fila, como o cliente que está no início da fileira e esperou por mais tempo.

Filas - 3


server
queue of customers
0
1
2
new arrival
at the end
enqueue
3
0
1
2
3
new arrival
at the end
enqueue
4
0
1
2
3
4
first in line
leaves queue
dequeue
0
0
1
2
3
4
next in line
leaves queue
dequeue
1
1
2
3
4

Atributos de uma fila

- Podemos implementar uma fila de no máximo `n-1` elementos usando um arranjo `Q[1 ... n]`.
- A fila tem um atributo `Q.início` que indexa ou aponta para seu início.
- O atributo `Q.fim` indexa a próxima posição na qual um elemento recém-chegado será inserido na fila.
- Os elementos na fila estão nas posições `Q.início`, `Q.início + 1`,..., `Q.fim - 1`, onde "retornamos", no sentido de que a posição 1 segue imediatamente a posição n em uma ordem circular.



Atributos de uma fila - 2

- Quando `Q.início = Q.fim`, a fila está vazia.
- Inicialmente, temos `Q.início = Q.fim = 1`.
- Se tentarmos desenfileirar um elemento de uma fila vazia, a fila sofre perda de dígitos.
- Quando `Q.início = Q.fim + 1` ou simultaneamente `Q.início= 1` e `Q.fim =Q.comprimento`, a fila está cheia e, se tentarmos enfileirar um elemento, a fila sofre estouro.

Pseudocódigo

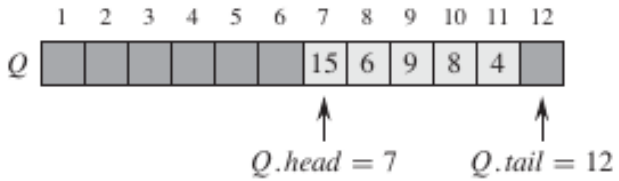
- Em nossos procedimentos `ENQUEUE` e `DEQUEUE`, omitimos a verificação de erros de estouro negativo e estouro.
- O pseudocódigo considera que `n = Q.comprimento`.

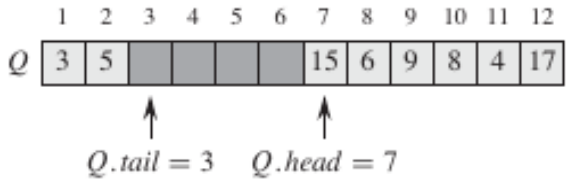
```
ENQUEUE(Q, x)
1  Q[Q.fim] = x
2  if Q.fim = Q.comprimento
3     Q.fim = 1
4  else Q.fim = Q.fim + 1
```

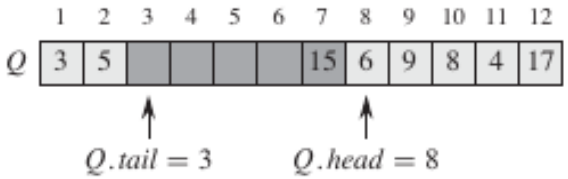
```
DEQUEUE(Q)
1  x = Q[Q.início]
2  if Q.início == Q.comprimento
3     Q.início = 1
4  else Q.início = Q.início + 1
5  return x
```

Exemplo

- (a)
  - A fila tem cinco elementos, nas posições `Q[ 7 ... 11]`.
- (b)
  - A configuração da fila após as chamadas `ENQUEUE(Q,17)`, `ENQUEUE(Q,3)` e `ENQUEUE(Q,5)`.
- (c)
  - A configuração da fila após a chamada `DEQUEUE(Q)` retornar o valor de chave 15 que se encontrava anteriormente no início da fila.
  - O novo início tem chave 6.







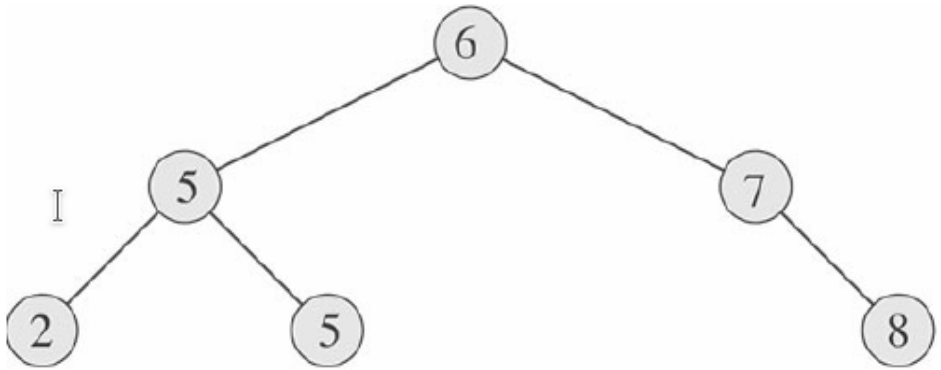
Aplicações de filas:

- A fila é usada quando as coisas não têm de ser processadas imediatamente, mas têm de ser processadas na ordem First In First Out.
- Esta propriedade de fila também a torna útil nos seguintes tipos de cenários.
  - Quando um recurso é compartilhado entre vários consumidores. Exemplos incluem programação de CPU, Programação de Disco.
  - Quando os dados são transferidos de forma assíncrona (dados não necessariamente recebidos ao mesmo ritmo que enviados) entre dois processos.
    Exemplos incluem IO Buffers, pipes, arquivo IO, etc.

Árvores de busca binária

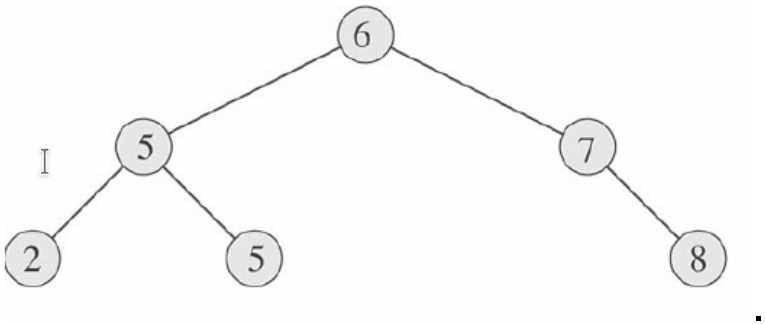
Árvore de busca binária

- Uma árvore de busca binária é organizada, como o nome sugere, em uma árvore binária.
- Podemos representar tal árvore por uma estrutura de dados ligada, na qual cada nó é um objeto.
- Além de uma chave e de dados satélites, cada nó contém atributos `esquerda`, `direita` e `p`, que apontam para os nós correspondentes ao seu filho à esquerda, ao seu filho à direita e ao seu pai, respectivamente.
- Se um filho ou o pai estiver ausente, o atributo adequado contém o valor `NIL`.
- O nó raiz é o único nó na árvore cujo pai é `NIL`.

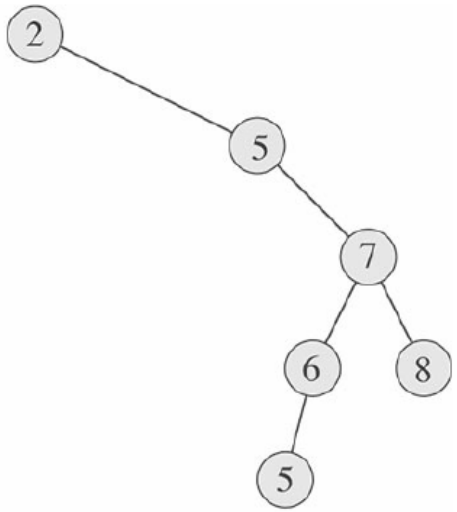


Propriedade de árvore de busca binária

- As chaves em uma árvore de busca binária são sempre armazenadas de modo a satisfazer a propriedade de árvore de busca binária:
  - Seja `x` um nó em uma árvore de busca binária.
  - Se `y` é um nó na subárvore esquerda de `x`, então `y.chave` $\leq$ `x.chave`.
  - Se `y` é um nó na subárvore direita de `x`, então `x.chave` $\geq$ `y.chave`.



Uma árvore de busca binária com seis nós e altura 2.



Uma árvore de busca binária menos eficiente, com altura 4, que contém as mesmas chaves.

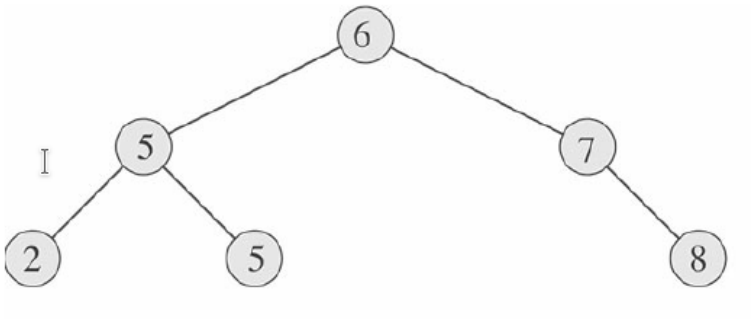
Imprimir todas as chaves

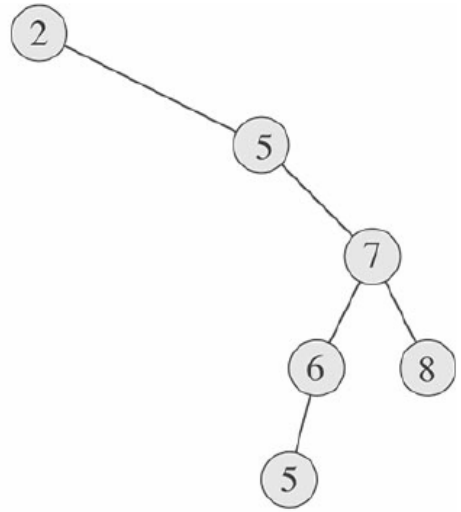
- A propriedade de árvore de busca binária nos permite imprimir todas as chaves em uma árvore de busca binária em sequência ordenada por meio de um simples algoritmo recursivo, denominado `percurso de árvore em in-ordem`.
- Esse algoritmo tem tal nome porque imprime a chave da raiz de uma subárvore entre a impressão dos valores em sua subárvore esquerda e a impressão dos valores em sua subárvore direita.
- Para usar o procedimento a seguir com o objetivo de imprimir todos os elementos em uma árvore de busca binária `T`, chamamos `INORDER-TREE-WALK(T.raiz)`.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.esquerda)
3      print x.chave
4      INORDER-TREE-WALK(x.direita)
```

Exemplo

- Como exemplo, o percurso de árvore em ordem imprime as chaves em cada uma das duas árvores de busca binária na ordem 2, 5, 5, 6, 7, 8.
- A correção do algoritmo decorre por indução diretamente da propriedade de árvore de busca binária.





Consultas em uma árvore de busca binária

- Frequentemente precisamos procurar uma chave armazenada em uma árvore de busca binária.
- Além da operação `SEARCH`, árvores de busca binária podem suportar as consultas
  - `MINIMUM`
  - `MAXIMUM`
  - `SUCCESSOR`
  - `PREDECESSOR`

Buscas

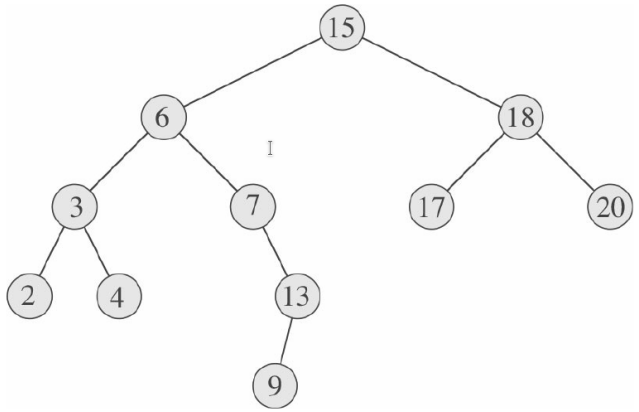
- Usamos o procedimento a seguir para procurar um nó com determinada chave em uma árvore de busca binária.
- Dado um ponteiro para a raiz da árvore e uma chave `k`, `TREE-SEARCH` retorna um ponteiro para um nó com chave `k`, se existir algum; caso contrário, ele retorna `NIL`.

```
ITERATIVE-TREE-SEARCH(x, k)
1   while x ≠ NIL e k ≠ x.chave
2       if k < x.chave
3           x = x.esquerda
4       else x = x.direita
5   return x
```

.

Buscas - 2

- O procedimento começa sua busca na raiz e traça um caminho simples descendo a árvore.
- Para cada nó `x` que encontra, ele compara a chave `k` com a `x.chave`.
- Se as duas chaves são iguais, a busca termina.
- Se `k` é menor que `x.chave`, a busca continua na subárvore esquerda de `x`, já que a propriedade de árvore de busca binária implica que `k` não poderia estar armazenada na subárvore direita.
- Simetricamente, se `k` é maior que `x.chave`, a busca continua na subárvore direita.



Para procurar a chave 13 na árvore, seguimos o caminho 15 - 6 - 7 - 13 partindo da raiz.

Mínimo

- Sempre podemos encontrar um elemento em uma árvore de busca binária cuja chave é um mínimo seguindo ponteiros de filhos da esquerda desde a raiz até encontrarmos um valor `NIL`.
- O procedimento a seguir, retorna um ponteiro para o elemento mínimo na subárvore enraizada em um nó `x` dado.

```
TREE-MINIMUM(x)
1  while x.esquerda ≠ NIL
2      x = x.esquerda
3  return x
```

Mínimo - 2

- A propriedade de árvore de busca binária garante que `TREE-MINIMUM` é correto.
- Se um nó `x` não tem nenhuma subárvore esquerda, então, visto que toda chave na subárvore direita de `x` é no mínimo tão grande quanto `x.chave`, a chave mínima na subárvore enraizada em `x` é `x.chave`.
- Se o nó `x` tem uma subárvore esquerda, então, visto que nenhuma chave na subárvore direita é menor que `x.chave` e toda chave na subárvore esquerda não é maior que `x.chave`, a chave mínima na subárvore enraizada em `x` pode ser encontrada na subárvore enraizada em `x.esquerda`.

Máximo

- O pseudocódigo para `TREE-MAXIMUM` é simétrico.

```
TREE-MAXIMUM(x)
1   while x.direita ≠ NIL
2       x = x.direita
3   return x
```

Sucessor e predecessor

- Dado um nó em uma árvore de busca binária, às vezes, precisamos encontrar seu sucessor na sequência ordenada determinada por um percurso de árvore em ordem.
- Se todas as chaves são distintas, o sucessor de um nó `x` é o nó com a menor chave maior que `chave[x]`.
- A estrutura de uma árvore de busca binária nos permite determinar o sucessor de um nó sem sequer comparar chaves.
- O procedimento a seguir, retorna o sucessor de um nó `x` em uma árvore de busca binária se ele existir, e `NIL` se `x` tem a maior chave na árvore.

```
TREE-SUCCESSOR(x)
1  if x.direita ≠ NIL
2     return TREE-MINIMUM(x.direita)
3  y = p[x]
4  while y ≠ NIL e x = y.direita
5     x = y
6     y = y.p
7  return y
```

Sucessor e predecessor - 2

- Subdividimos o código para `TREE-SUCCESSOR` em dois casos.
- Se a subárvore direita do nó `x` for não vazia, então o sucessor de `x` é exatamente o nó da extrema esquerda na subárvore direita de `x`, que encontramos na linha 2 chamando `TREE-MINIMUM(x.direita)`.
- Por outro lado, se a subárvore direita do nó `x` é vazia e `x` tem um sucessor `y`, então `y` é o ancestral mais baixo de `x` cujo filho à esquerda é também um ancestral de `x`.
- Para encontrar `y`, simplesmente subimos a árvore desde `x` até encontrarmos um nó que seja o filho à esquerda de seu pai.

```
TREE-SUCCESSOR(x)
1   if x.direita ≠ NIL
2       return TREE-MINIMUM(x.direita)
3   y = p[x]
4   while y ≠ NIL e x = y.direita
5       x = y
6       y = y.p
7   return y
```

Inserção e eliminação

- As operações de inserção e eliminação provocam mudanças no conjunto dinâmico representado por uma árvore de busca binária.
- A estrutura de dados deve ser modificada para refletir essa mudança, mas de tal modo que a propriedade de árvore de busca binária continue válida.
- Como veremos, modificar a árvore para inserir um novo elemento é uma operação relativamente direta, mas manipular a eliminação é um pouco mais complicado.

Inserção

- Para inserir um novo valor `v` em uma árvore de busca binária `T`, utilizamos o procedimento `TREE-INSERT`.
- O procedimento toma um nó `z` para o qual `z.chave = v`, `z.esquerda = NIL` e `z.direita = NIL`, e modifica `T` e alguns dos atributos de `z` de modo tal que insere `z` em uma posição adequada na árvore.

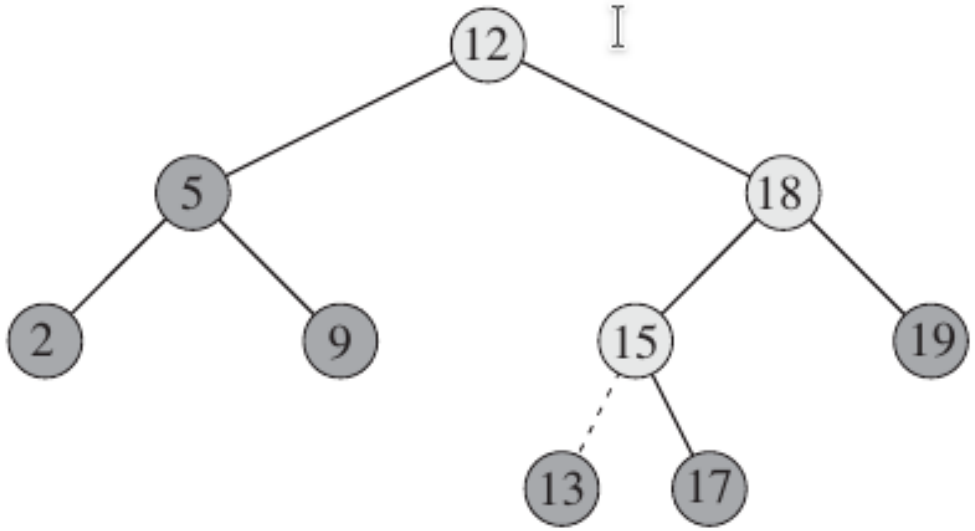
```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.raiz
3   while x ≠ NIL
4       y = x
5       if z.chave < x.chave
6               x = x.esquerda
7       else x = x.direita
8   z.p = y
9   if y = NIL
10      T.raiz = z                  //a árvore T era vazia
11  else if z.chave < y.chave
12      y.esquerda = z
13  else y.direita = z
```

Inserção - 2

- `TREE-INSERT` começa na raiz da árvore e o ponteiro `x` e traça um caminho simples descendente procurando um `NIL` para substituir pelo item de entrada `z`.
- O procedimento mantém o ponteiro acompanhante `y` como o pai de `x`.
- Após a inicialização, o laço `while` faz com que esses dois ponteiros se desloquem para baixo na árvore, indo para a esquerda ou a direita, dependendo da comparação de `z.chave` com `x.chave`, até `x` tornar-se `NIL`.
- Esse `NIL` ocupa a posição na qual desejamos colocar o item de entrada `z`.
- Precisamos do ponteiro acompanhante `y` porque, até encontrarmos o `NIL` ao qual `z` pertence, a busca desenvolveu-se sempre uma etapa à frente do nó que precisa ser mudado.

Exemplo

- Inserção de um item com chave 13 em uma árvore de busca binária.
- Os nós sombreados em tom mais claro indicam o caminho simples da raiz até a posição em que o item é inserido.
- A linha tracejada indica a ligação que é acrescentada à árvore para inserir o item.

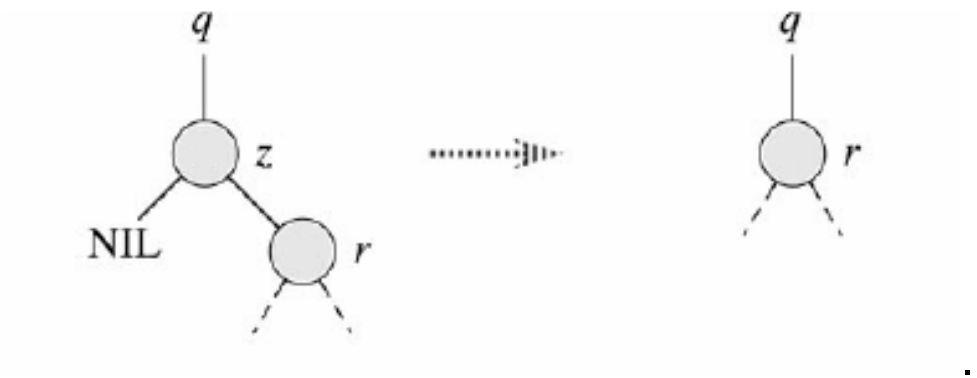


Eliminação

- A estratégia global para eliminar um nó z de uma árvore de busca binária T tem três casos básicos, mas, como veremos, um deles é um pouco complicado.
  - Se z não tem nenhum filho, então simplesmente o removemos modificando seu pai de modo a substituir z por NIL como seu filho.
  - Se o nó tem apenas um filho, então elevamos esse filho para que ocupe a posição de z na árvore modificando o pai de z de modo a substituir z pelo filho de z.
  - Se z tiver dois filhos, encontramos o sucessor de z, y, que deve estar na subárvore direita de z, e obrigamos y a tomar a posição de z na árvore. O resto da subárvore direita original de z torna-se a nova subárvore direita de y, e a subárvore esquerda de z torna-se a nova subárvore esquerda de y.
    Esse é o caso complicado porque, como veremos, o fato de y ser ou não o filho à direita de z é importante.

Eliminação - 2

- O procedimento para eliminar um dado nó z de uma árvore de busca binária T toma como argumentos ponteiros para T e z.
- Esse procedimento organiza seus casos de um modo um pouco diferente dos três casos descritos, já que considera quatro casos.
- Caso 1:
  - Se z não tiver nenhum filho à esquerda, substituímos z por seu filho à direita, que pode ser ou não NIL.
  - Quando o filho à direita de z é NIL, esse caso trata da situação na qual z não tem nenhum filho.
  - Quando o filho à direita de z é não NIL, esse caso manipula a situação na qual z tem somente um filho, que é o seu filho à direita.

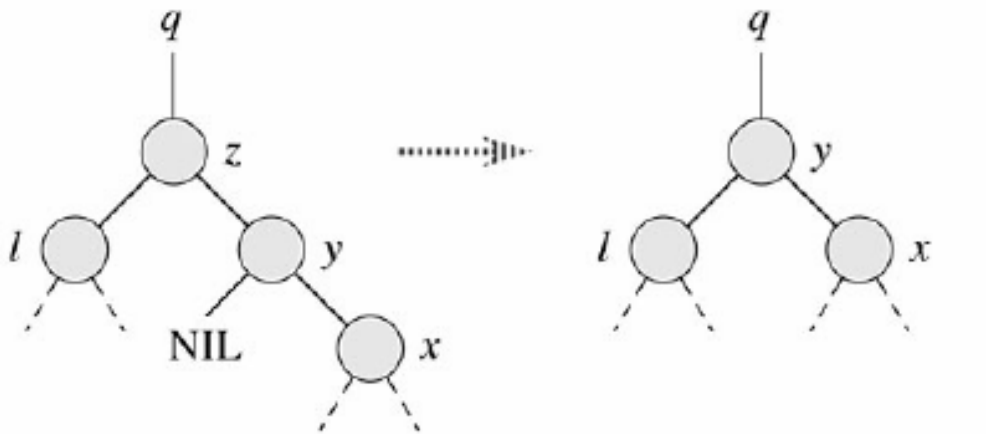


Eliminação - 3

- Caso 2:
  - Se z tiver apenas um filho, que é o seu filho à esquerda, substituímos z por seu filho à esquerda.

Eliminação - 4

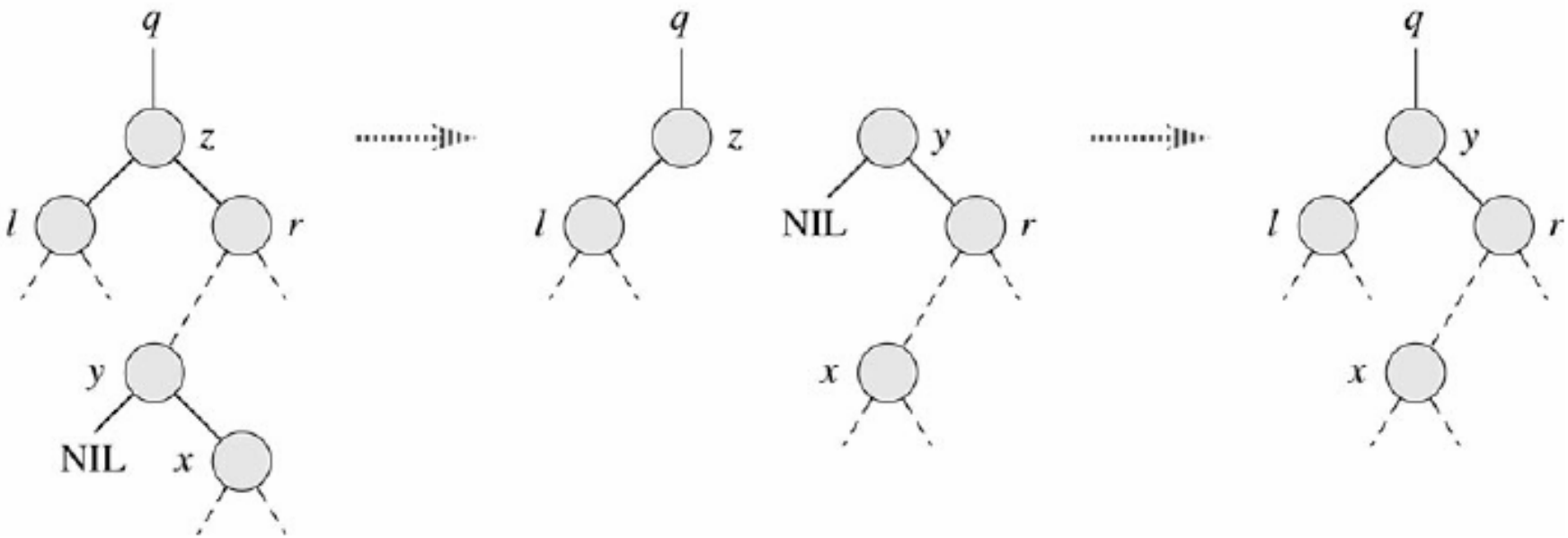
- Caso 3-a:
  - z tem um filho à esquerda e também um filho à direita.
  - Encontramos o sucessor de z, y, que está na subárvore direita de z e não tem nenhum filho à esquerda.
  - Queremos recortar y de sua localização atual e fazê-lo substituir z na árvore.
  - Se y é o filho à direita de z, substituímos z por y, deixando o filho à direita de y sozinho.



.

Eliminação - 5

- Caso 3-b:
  - Caso contrário, y encontra-se dentro da subárvore direita de z, mas não é o filho à direita de z.
  - Nesse caso, primeiro substituímos y por seu próprio filho à direita e depois substituímos z por y.



Transplant

- Para movimentar subárvores dentro da árvore de busca binária, definimos uma subrotina `TRANSPLANT`, que substitui uma subárvore como um filho de seu pai por uma outra subárvore.
- Quando `TRANSPLANT` substitui a subárvore enraizada no nó `u` pela subárvore enraizada no nó `v`, o pai do nó `u` torna-se o pai do nó `v`, e o pai de `u` acaba ficando com `v` como seu filho adequado.

```
TRANSPLANT (T, u, v)

1  if u.p == NIL
2      T.raiz = v
3  elseif u == u.p.esquerda
4      u.p. esquerda = v
5  else u.p.direita = v
6  if v ≠ NIL
7      v.p = u.p
```

Transplant - 2

- As linhas 1-2 tratam do caso no qual `u` é a raiz de `T`.
- Caso contrário, `u` é um filho à esquerda ou um filho à direita de seu pai.
- As linhas 3-4 encarregam-se de atualizar `u.p.direita` se `u` é um filho à direita, e a linha 5 atualiza `u.p.esquerda` se `u` é um filho à esquerda.
- Permitimos que `v` seja `NIL`, e as linhas 6-7 atualizam `v.p` se `v` é não `NIL`.
- Observe que `TRANSPLANT` não tenta atualizar `v.esquerda` e `v.direita`; fazer ou não fazer isso é responsabilidade do chamador de `TRANSPLANT`.

Eliminação - 6

- Com o procedimento `TRANSPLANT` em mãos, apresentamos a seguir o procedimento que elimina o nó `z` da árvore de busca binária `T`:

```
TREE-DELETE (T,z)
1   if z.esquerda == NIL
2       TRANSPLANT (T, z, z.direita)
3   elseif z.direita == NIL
4       TRANSPLANT (T, z, z.esquerda)
5   else y = TREE-MINIMUM (z.direita)
6       if y.p ≠ z
7               TRANSPLANT (T, y, y.direita)
8               y.direita = z.direita
9               y.direita.p = y
10      TRANSPLANT (T, z, y)
11      y.esquerda = z.esquerda
12      y.esquerda.p = y
```

Eliminação - 7

- O procedimento `TREE-DELETE` executa os quatro casos como descrevemos a seguir.
- As linhas 1-2 tratam do caso no qual o nó `z` não tem nenhum filho à esquerda.
- As linhas 3-4 tratam do caso no qual `z` tem um filho à esquerda mas nenhum filho à direita.
- As linhas 5-12 lidam com os dois casos restantes, nos quais `z` tem dois filhos.
  - A linha 5 encontra o nó `y`, que é o sucessor de `z`.
    - Como `z` tem uma subárvore direita não vazia, seu sucessor deve ser o nó nessa subárvore que tenha a menor chave; daí a chamada a `TREE-MINIMUM (z.direita)`.
    - Como observamos antes, `y` não tem nenhum filho à esquerda.
    - Queremos recortar `y` de sua localização atual, e ele deve substituir `z` na árvore.

Eliminação - 8

- Se y é o filho à direita de z, então as linhas 10-12 substituem z como um filho de seu pai por y e substituí o filho à esquerda de y pelo filho à esquerda de z.
- Se y não é o filho à direita de z , as linhas 7-9 substituem y como um filho de seu pai pelo filho à direita de y e transforma o filho à direita de z em filho à direita de y, e então as linhas 10-12 substituem z como um filho de seu pai por y e substituem o filho à esquerda de y pelo filho à esquerda de z.

Tabelas de espalhamento

Exemplo: sistema de registros

- Suponha que queremos projetar um sistema de armazenamento de registros de funcionários usando números de telefone. E queremos que as consultas seguintes sejam realizadas de forma eficiente:
  - Inserir um número de telefone e as informações correspondentes.
  - Pesquisar um número de telefone e buscar as informações.
  - Eliminar um número de telefone e as informações relacionadas.
- Podemos pensar em usar as seguintes estruturas de dados para manter as informações sobre diferentes números de telefone.
  - Arranjo de números de telefone e registros.
  - Lista ligadas de números de telefone e registros.
  - Árvore de busca binária balanceada com números de telefone como chaves.
  - Tabela de acesso direto.
- Para arranjos e listas vinculadas, precisamos pesquisar de forma linear, o que pode ser dispendioso na prática.
- Com a árvore de busca binária, obtemos tempos moderados de busca, inserção e exclusão.

Exemplo: sistema de registros - 2

- Outra solução que se pode pensar é usar uma tabela de acesso direto onde fazemos uma grande matriz e usamos números de telefone como índice na matriz.
- Uma entrada na matriz é `NIL` se o número de telefone não estiver presente, caso contrário, a entrada na matriz armazena o ponteiro para os registros correspondentes ao número de telefone.
- Por exemplo, para inserir um número de telefone, criamos um registro com detalhes de um determinado número de telefone, usamos o número de telefone como índice e armazenamos o ponteiro para o registro criado na tabela.

Tabelas de espalhamento

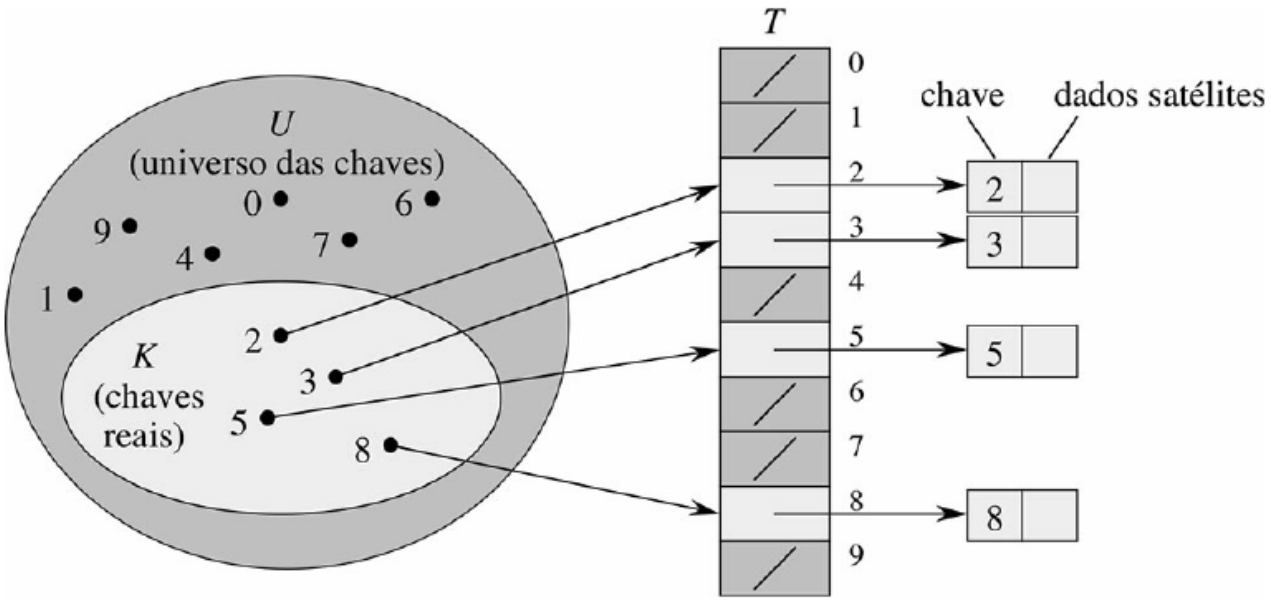
- Muitas aplicações exigem um conjunto dinâmico que suporte somente as operações de dicionário `INSERT` , `SEARCH` e `DELETE`.
- Por exemplo, um compilador que traduz uma linguagem de programação mantém uma tabela de símbolos na qual as chaves de elementos são cadeias de caracteres arbitrários que correspondem a identificadores na linguagem.
- Uma tabela de **espalhamento** ou **hashing** é uma estrutura de dados eficaz para implementar dicionários.

Tabelas de endereço direto

- O endereçamento direto é uma técnica simples que funciona bem quando o universo U de chaves é razoavelmente pequeno.
- Suponha que uma aplicação necessite de um conjunto dinâmico no qual cada elemento tem uma chave extraída do universo $U = \{0, 1, ..., m - 1\}$, onde m não é muito grande.
- Consideramos que não há dois elementos com a mesma chave.

Tabelas de endereço direto - 2

- Para representar o conjunto dinâmico, usamos um arranjo, ou uma tabela de endereços diretos, denotada por $T[0..m-1]$, na qual cada posição, ou lacuna, corresponde a uma chave no universo U.
- Na figura a posição k aponta para um elemento no conjunto com chave k.
- Se o conjunto não contém nenhum elemento com chave k, então $T[k] = NIL$.



Tabelas de endereço direto - 3

- A implementação das operações de dicionário é trivial.

```
DIRECT-ADDRESS-SEARCH(T, k)
1   return T[k] = x

DIRECT-ADDRESS-INSERT(T, x)
1   T[x.chave] = x

DIRECT-ADDRESS-DELETE(T, x)
1   T[x.chave] = NIL
```

Implementação

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000

// Since array is global, it is initialized as 0.
bool has[MAX + 1][2];

// Searching if X is Present in the given array or not.
bool search(int X)
{
    if (X >= 0) {
        if (has[X][0] == 1)
            return true;
        else
            return false;
    }
    // if X is negative take the absolute value of X.
    X = abs(X);
    if (has[X][1] == 1)
        return true;
    return false;
}
```

Implementação

```
void insert(int a[], int n)
{
    for (int i = 0; i < n; i++) {
        if (a[i] >= 0)
            has[a[i]][0] = 1;
       else
            has[abs(a[i])][1] = 1;
    }
}
// Driver code
int main()
{
    int a[] = { -1, 9, -5, -8, -5, -2 };
    int n = sizeof(a)/sizeof(a[0]);
    insert(a, n);
    int X = -5;
    if (search(X) == true)
       cout << "Present";
    else
       cout << "Not␣Present";
    return 0;
}
```

Tabelas de espalhamento

- O aspecto negativo do endereçamento direto é óbvio: se o universo U é grande, armazenar uma tabela T de tamanho $|U|$ pode ser impraticável ou mesmo impossível, dada a memória disponível em um computador típico.
- Além disso, o conjunto K de chaves realmente armazenadas pode ser tão pequeno em relação a U que grande parte do espaço alocado para T seria desperdiçada.
- Quando o conjunto K de chaves armazenadas em um dicionário é muito menor que o universo U de todas as chaves possíveis, uma tabela de espalhamento requer armazenamento muito menor que uma tabela de endereços diretos.
- Especificamente, podemos reduzir o requisito de armazenamento a $(|K|)$ e ao mesmo tempo manter o benefício de procurar um elemento na tabela em um tempo curto.

Exemplo: sistema de registros - 3

- Uma função hash converte um determinado número de telefone grande para um pequeno valor inteiro prático.
- O valor inteiro mapeado é usado como um índice na tabela de hash.
- Em termos simples, uma função hash mapeia um número grande ou string para um número inteiro pequeno que pode ser usado como índice em uma tabela hash.
- Uma boa função de hash deve ter as seguintes propriedades
  - Eficientemente calculável.
  - Deve distribuir uniformemente as chaves (cada posição de tabela igualmente provável para cada chave)
- Por exemplo, para números de telefone, uma função de hash ruim é tomar os três primeiros dígitos. Uma função melhor é considerar os três últimos dígitos. Observe que esta pode não ser a melhor função de hash. Pode haver maneiras melhores.

Função hash

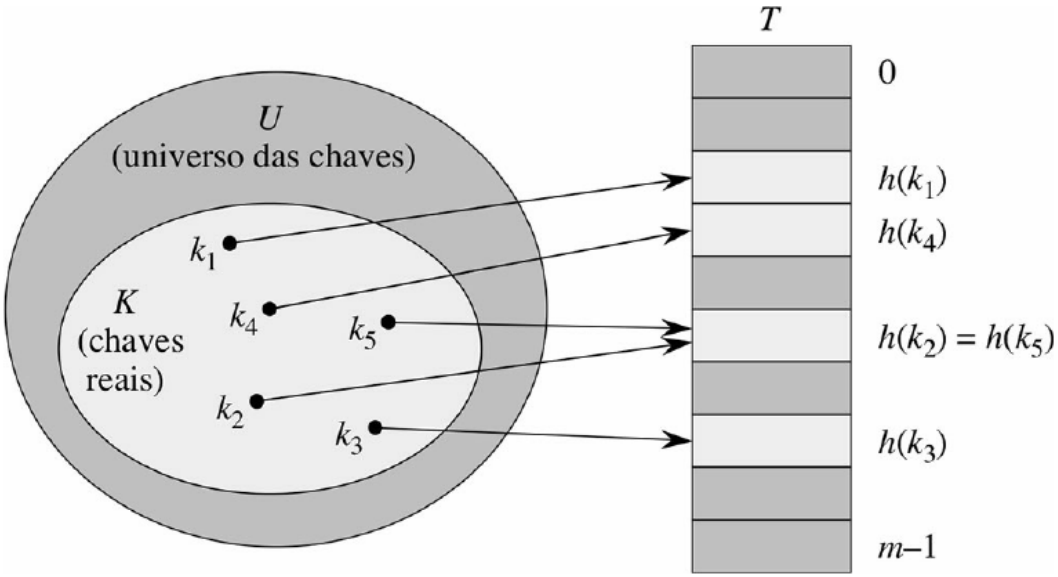
- Com endereçamento direto, um elemento com a chave k é armazenado na posição k.
- Com **hash**, esse elemento é armazenado na posição $h(k)$; isto é, usamos uma função hash h para calcular a posição pela chave k.
- Aqui, h mapeia o universo U de chaves para as posições de uma tabela de espalhamento $T[0 \ldots m-1]$:

$$h : U \to \{1, \ldots, m-1\},$$

  onde o tamanho m da tabela de espalhamento normalmente é muito menor que $|U|$.
- Dizemos que um elemento com a chave k se espalha (hashes) até a posição $h(k)$; dizemos também que $h(k)$ é o valor hash da chave k.

Função hash - 2

- A função hash reduz a faixa de índices do arranjo e, consequentemente, o tamanho do arranjo.
- Em vez de ter tamanho $|U|$, o arranjo pode ter tamanho m.



- Porém há um revés: após o hash, duas chaves podem ser mapeadas para a mesma posição.
- Chamamos essa situação de colisão. Felizmente, existem técnicas eficazes para resolver o conflito criado por colisões.

Funções hash

- Uma boa função hash satisfaz a premissa do hashing uniforme simples: cada chave tem igual probabilidade de passar para qualquer das m posições por uma operação de hash, independentemente da posição que qualquer outra chave ocupou após o hash.
- Infelizmente, normalmente não temos nenhum meio de verificar essa condição, já que raramente conhecemos a distribuição de probabilidade da qual as chaves são extraídas.

Funções hash - 2

- Na prática, muitas vezes, podemos usar técnicas heurísticas para criar uma função hash que funciona bem.
- Informações qualitativas sobre a distribuição de chaves podem ser úteis nesse processo de projeto.
- Por exemplo, considere a tabela de símbolos de um compilador, na qual as chaves são cadeias de caracteres que representam identificadores em um programa.
- Símbolos estreitamente relacionados, como `pt` e `pts`, frequentemente ocorrem no mesmo programa.
- Uma boa função hash minimizaria a chance de tais variações passarem para a mesma posição após o hashing.

Funções hash - 3

- Uma boa abordagem deriva o valor hash de um modo que esperamos seja independente de quaisquer padrões que possam existir nos dados.
- Por exemplo, o "método de divisão" calcula o valor hash como o resto quando a chave é dividida por um número primo especificado.
- Esse método frequentemente dá bons resultados, desde que que escolhamos um número primo que não esteja relacionado com quaisquer padrões na distribuição de chaves.

Interpretação de chaves como números naturais

- A maior parte das funções hash considera como universo de chaves o conjunto $U = \{0, 1, 2, ...\}$ de números naturais.
- Assim, se as chaves não forem números naturais, temos de encontrar um modo de interpretá-las como números naturais.
- Por exemplo, podemos interpretar uma cadeia de caracteres como um inteiro expresso em uma notação de raiz adequada.
- Assim, poderíamos interpretar o identificador `pt` como o par de inteiros decimais (112, 116), já que $p = 112$ e $t = 116$ no conjunto de caracteres ASCII
- Então, expresso como um inteiro de raiz 128, `pt` se torna $(112.128) + 116 = 14.452$.
- No contexto de uma determinada aplicação, normalmente podemos elaborar algum método para interpretar cada chave como um número natural (possivelmente grande).
- No que vem a seguir, supomos que as chaves são números naturais.

O método de divisão

- No método de divisão para criar funções hash, mapeamos uma chave k para uma de m posições, tomando o resto da divisão de k por m.
- Isto é, a função hash é

$$h(k) = k \mod m$$

- Por exemplo, se a tabela de espalhamento tem tamanho $m = 12$ e a chave é $k = 100$, então $h(k) = 4$.
- Visto que exige uma única operação de divisão, o hash por divisão é bastante rápido.

O método de divisão - 2

- Quando utilizamos o método de divisão, em geral evitamos certos valores de m.
- Muitas vezes, um primo não muito próximo de uma potência exata de 2 é uma boa escolha para m.
- Por exemplo, suponha que desejemos alocar uma tabela de espalhamento, com colisões resolvidas por encadeamento, para conter aproximadamente $n = 2.000$ cadeias de caracteres, onde um caractere tem oito bits.
- Não nos importamos de examinar uma média de três elementos em uma busca mal sucedida e, assim, alocamos uma tabela de espalhamento de tamanho $m \geq 701$.
- Pudemos escolher o número 701 porque é um primo próximo de $2.000/3$, mas não próximo de nenhuma potência de 2.
- Tratando cada chave k como um inteiro, nossa função hash seria

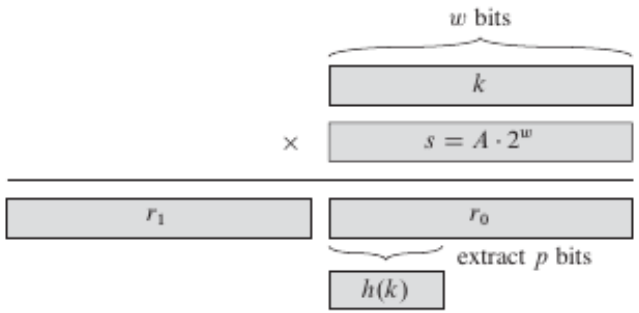
$$h(k) = k \mod 701$$

O método de multiplicação

- O método de multiplicação para criar funções hash funciona em duas etapas.
- Primeiro, multiplicamos a chave k por uma constante A na faixa $0 < A < 1$ e extraímos a parte fracionária de kA.
- Em seguida, multiplicamos esse valor por m e tomamos o piso do resultado. Resumindo, a função hash é:

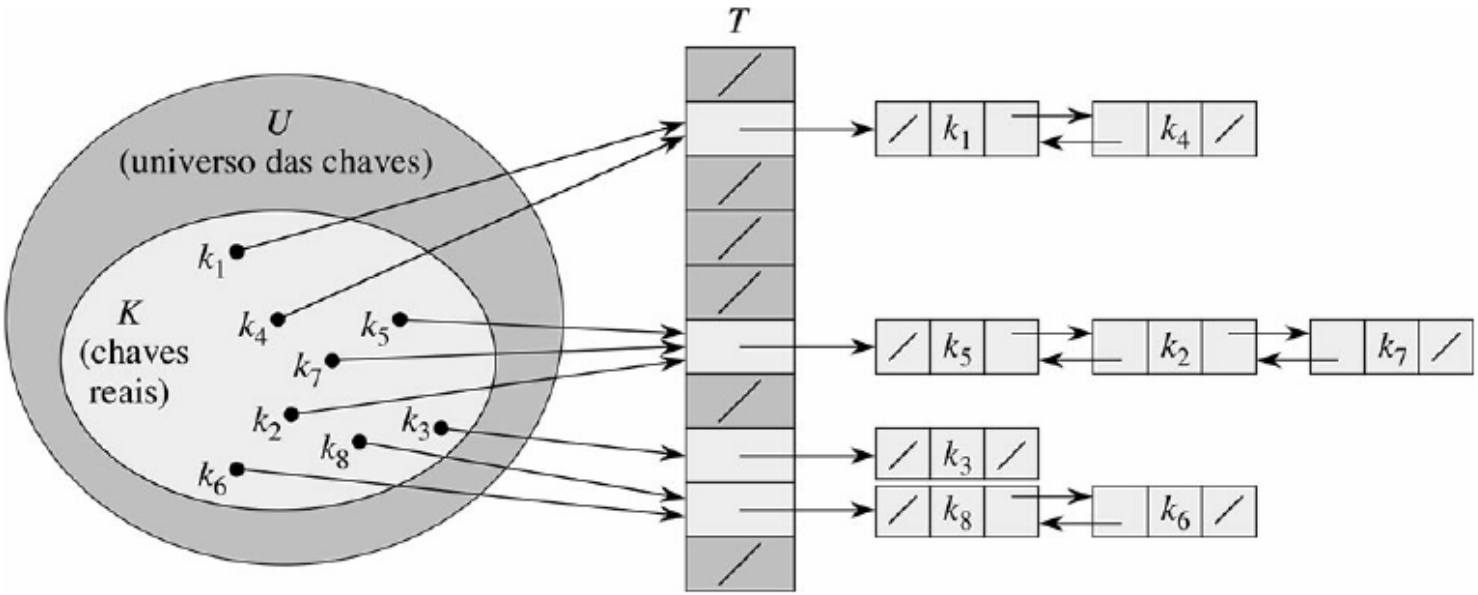
$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

- Uma vantagem do método de multiplicação é que o valor de m não é crítico.



Resolução de colisões por encadeamento

- No encadeamento, todos os elementos resultantes do hash vão para a mesma posição em uma lista ligada.
- A posição j contém um ponteiro para o início da lista de todos os elementos armazenados que, após o hash, foram para j; se não houver nenhum desses elementos, a posição j contém `NIL`.



Resolução de colisões por encadeamento - 2

- As operações de dicionário em uma tabela de espalhamento T são fáceis de implementar quando as colisões são resolvidas por encadeamento.

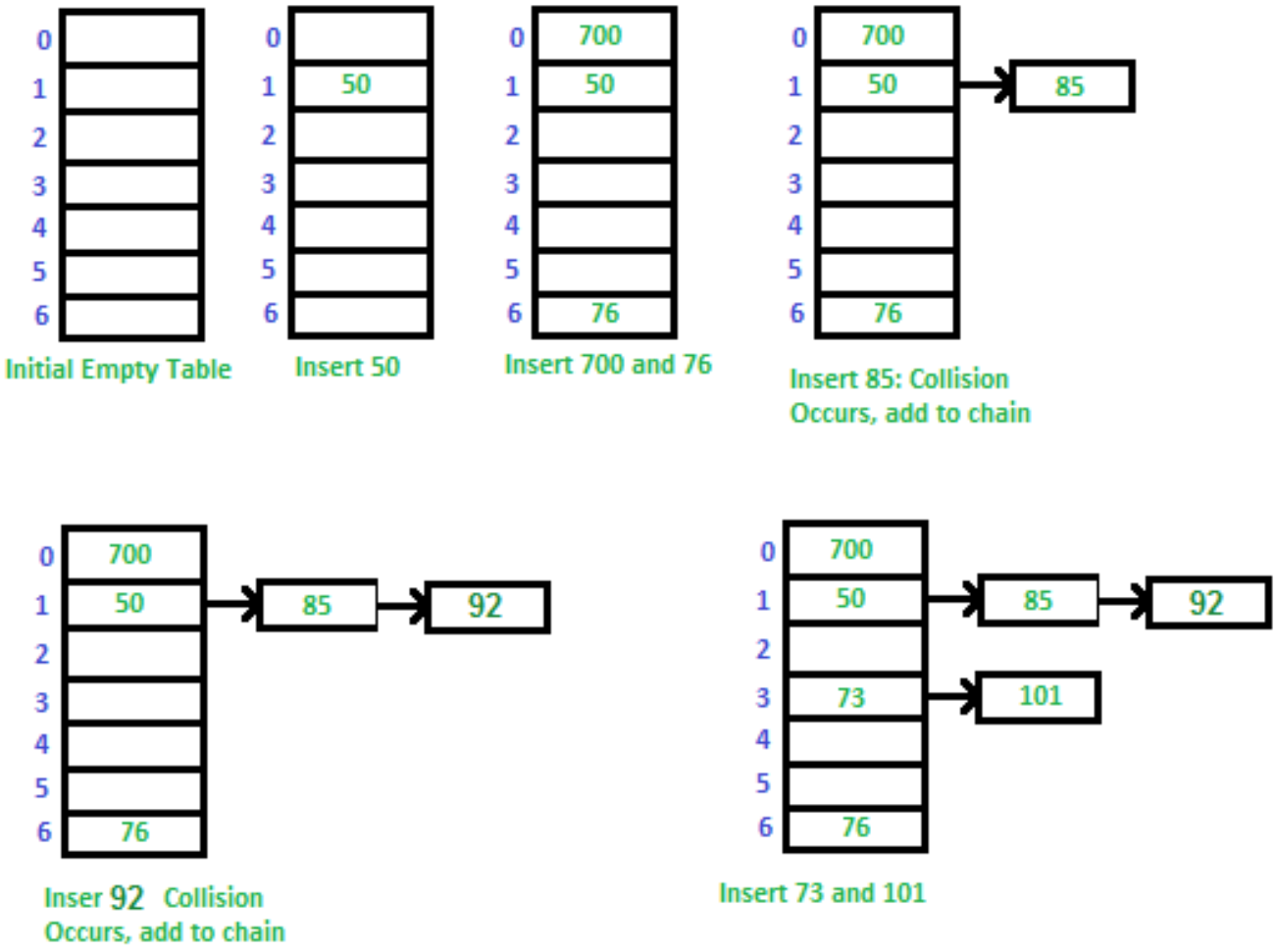
```
CHAINED-HASH-INSERT(T, x)
1   insere x no início da lista T[h(x.chave)]

CHAINED-HASH-SEARCH(T, k)
1   procura um elemento com a chave k na lista T[h(k)]

CHAINED-HASH-DELETE(T, x)
1   elimina x da lista T[h(x.chave)]
```

Exemplo

- O método de divisão com resolução de colisões por encadeamento.



Resolução de colisões por endereçamento aberto

- Em endereçamento aberto, todos os elementos ficam na própria tabela de espalhamento.
- Isto é, cada entrada da tabela contém um elemento do conjunto dinâmico ou NIL.
- Ao procurar um elemento, examinamos sistematicamente as posições da tabela até encontrar o elemento desejado ou até confirmar que o elemento não está na tabela.
- Diferentemente do encadeamento, não existe nenhuma lista e nenhum elemento armazenado fora da tabela.
- Assim, no endereçamento aberto, a tabela de espalhamento pode "ficar cheia", de tal forma que nenhuma inserção adicional pode ser feita.

Resolução de colisões por endereçamento aberto - 2

- A vantagem do endereçamento aberto é que ele evita por completo os ponteiros.
- Em vez de seguir ponteiros, calculamos a sequência de posições a examinar.
- A memória extra liberada por não armazenarmos ponteiros fornece à tabela de espalhamento um número maior de posições para a mesma quantidade de memória, o que produz potencialmente menor número de colisões e recuperação mais rápida.

Inserção com endereçamento aberto

- Para executar inserção usando endereçamento aberto, examinamos sucessivamente, ou sondamos, a tabela de espalhamento até encontrar uma posição vazia na qual inserir a chave.
- Em vez de ser fixa na ordem $0, 1, ..., m-1$, a sequência de posições sondadas **depende da chave que está sendo inserida**.
- Para determinar quais serão as posições a sondar, estendemos a função hash para incluir o número da sondagem (a partir de 0) como uma segunda entrada.
- Assim, a função hash se torna:

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

- Com endereçamento aberto, exigimos que, para toda chave k, a sequência de sondagem $\langle h(k, 0), h(k, 1), ..., h(k, m-1) \rangle$ seja uma permutação de $\langle 0, 1, ..., m-1 \rangle$, de modo que toda posição da tabela de espalhamento seja eventualmente considerada uma posição para uma nova chave, à medida que a tabela é preenchida.

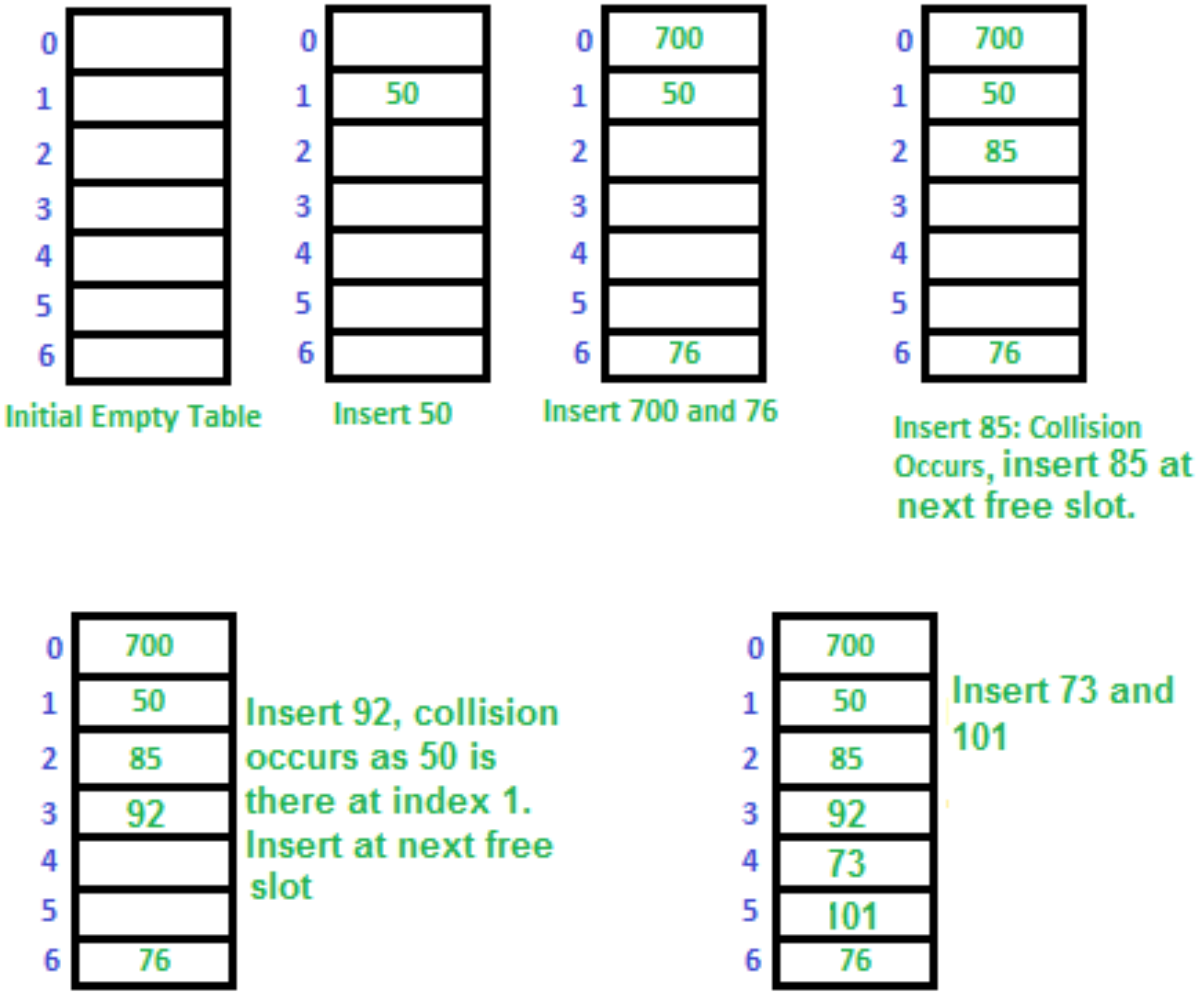
Inserção com endereçamento aberto - 2

- No pseudocódigo a seguir, supomos que os elementos na tabela de espalhamento `T` são chaves sem informações satélites; a chave `k` é idêntica ao elemento que contém a chave `k`.
- Cada posição contém uma chave ou `NIL` (se a posição estiver vazia).
- O procedimento `HASH-INSERT` tem como entrada uma tabela de espalhamento `T` e uma chave `k`.
- Devolve o número da posição onde armazena chave `k` ou sinaliza um erro porque a tabela de espalhamento já está cheia.

```
HASH-INSERT(T, k)
1 i = 0
2 repeat j = h(k, i)
3       if T[j] == NIL
4           T[j] = k
5           return j
6       else i = i + 1
7 until i == m
8 error "estouro da tabela"
```

Exemplo

- O método de divisão com resolução de colisões por encadeamento.



Busca com endereçamento aberto

- O algoritmo que procura a chave `k` sonda a mesma sequência de posições que o algoritmo de inserção examinou quando a chave `k` foi inserida.
- Portanto, a busca pode terminar (sem sucesso) quando encontra uma posição vazia, já que `k` teria sido inserido ali e não mais adiante em sua sequência de sondagem. (Esse argumento supõe que não há eliminação de chaves na tabela de espalhamento.)
- O procedimento `HASH-SEARCH` tem como entrada uma tabela de espalhamento `T` e um chave `k`, e devolve `j` se verificar que a posição `j` contém a chave `k`, ou `NIL` se a chave `k` não estiver presente na tabela.

```
HASH-SEARCH(T, k)
1 i = 0
2 repeat
3     j = h(k, i)
4       if T[j] == k
5           return j
6       i = i + 1
7 until T[j] == NIL ou i == m
8 return NIL
```

Eliminação com endereçamento aberto

- Eliminar algo em uma tabela de espalhamento de endereço aberto é difícil.
- Quando eliminamos uma chave da posição `i`, não podemos simplesmente marcar essa posição como vazia, nela armazenando `NIL`.
- Se fizéssemos isso, poderíamos não conseguir recuperar nenhuma chave `k` em cuja inserção tivéssemos sondado a posição `i` e verificado que ela estava ocupada.
- Podemos resolver esse problema marcando a posição nela armazenando o valor especial `DELETED` em vez de `NIL`.
- Então, modificaríamos o procedimento `HASH-INSERT` para tratar tal posição como se ela estivesse vazia, de modo a podermos inserir uma nova chave.
- Não é necessário modificar `HASH-SEARCH`, já que ele passará por valores `DELETED` enquanto estiver pesquisando.
- O encadeamento é mais comumente selecionado como uma técnica de resolução de colisões quando precisamos eliminar chaves.

Encadeamento

- O encadeamento é mais simples de implementar.
- No encadeamento, a tabela de Hash nunca se enche, podemos sempre adicionar mais elementos à cadeia.
- O encadeamento é menos sensível à função hash ou aos fatores de carga.
- O encadeamento é usado principalmente quando não se sabe quantas e com que frequência as chaves podem ser inseridas ou excluídas.
- O desempenho de cache do encadeamento não é bom, pois as chaves são armazenadas usando lista ligada.
- Desperdício de espaço (Algumas partes da tabela de hash no encadeamento nunca são usadas).
- O encadeamento utiliza espaço extra para links.

Endereçamento aberto

- O endereçamento aberto requer mais computação.
- Em endereçamento aberto, a tabela pode ficar cheia.
- O endereçamento aberto requer cuidado extra para evitar o agrupamento e o fator de carga.
- O endereçamento aberto é usado quando a frequência e o número de chaves é conhecido.
- O endereçamento aberto proporciona melhor desempenho de cache, uma vez que tudo é armazenado na mesma tabela.
- No endereçamento aberto, um slot pode ser usado mesmo que uma entrada não seja mapeada para ele.
- Sem links em endereçamento aberto.