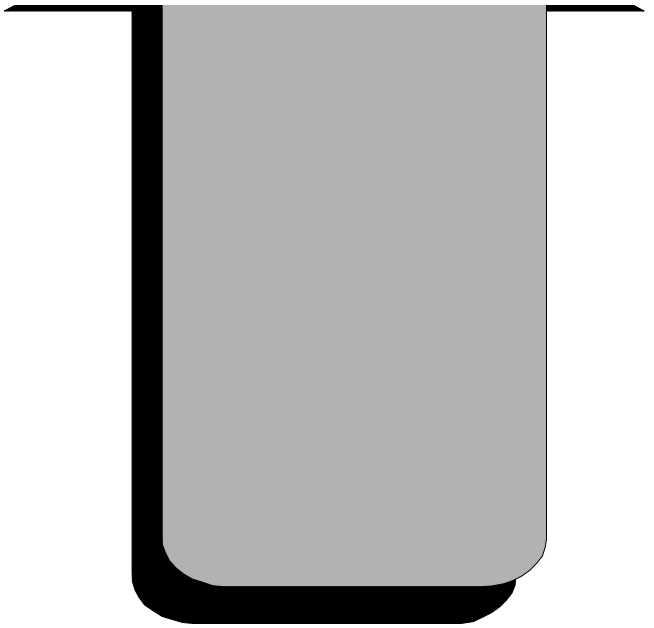
Pilhas

Organização

- Definição do ADT Pilha
- Especificação
  - Operações sobre o ADT Pilha, utilizando pré- e pós-condições
- Implementação
  - Estática (contígua)
  - Dinâmica (encadeada)

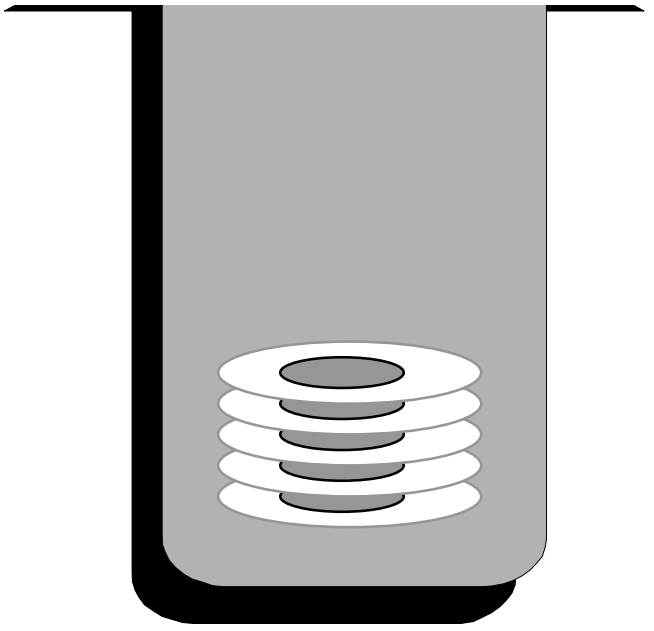
Definição

- Uma **pilha** (*stack*) é uma estrutura de dados na qual todas inserções e remoções de dados são efetuadas numa única extremidade, denominada **topo** (*top*) da pilha



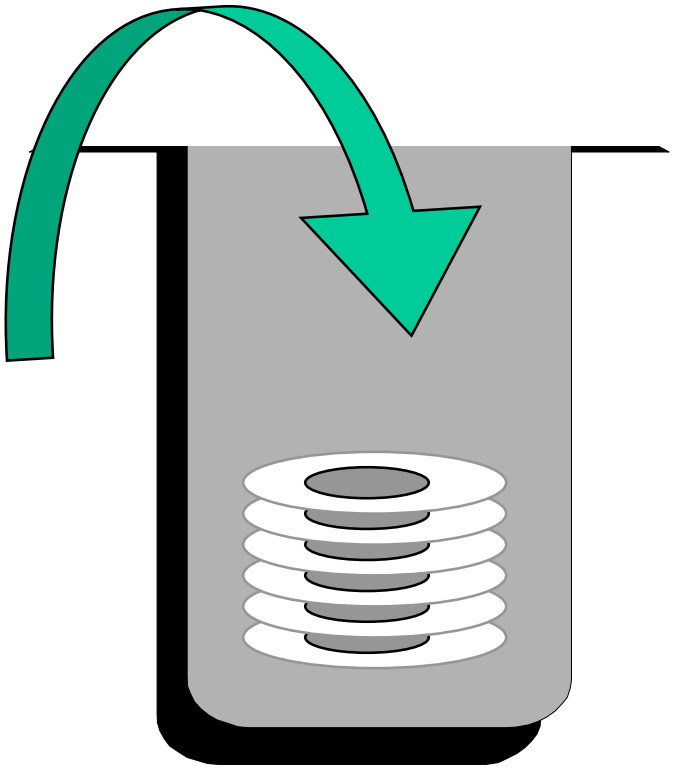
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo *self-service*



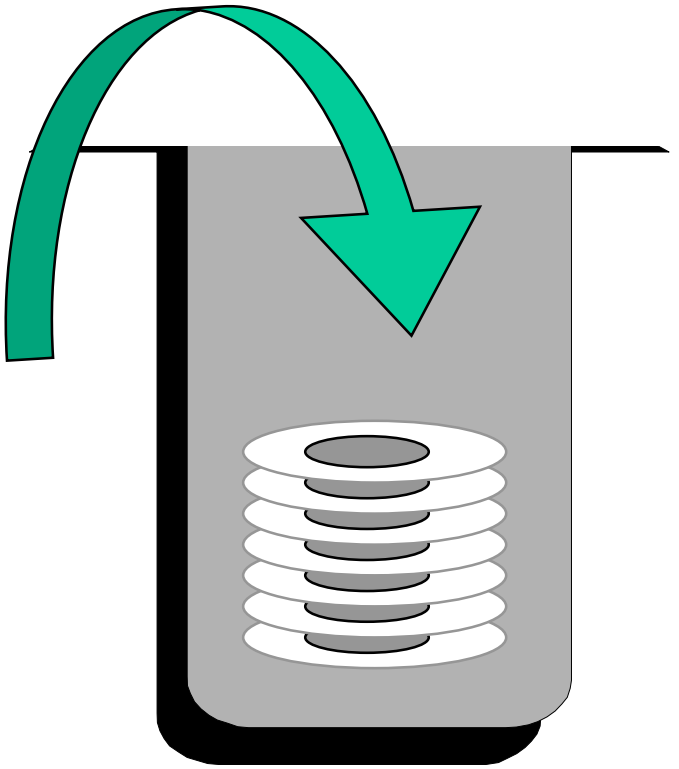
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo *self-service*
- Os pratos são colocados sempre no topo da pilha pelos empregados...



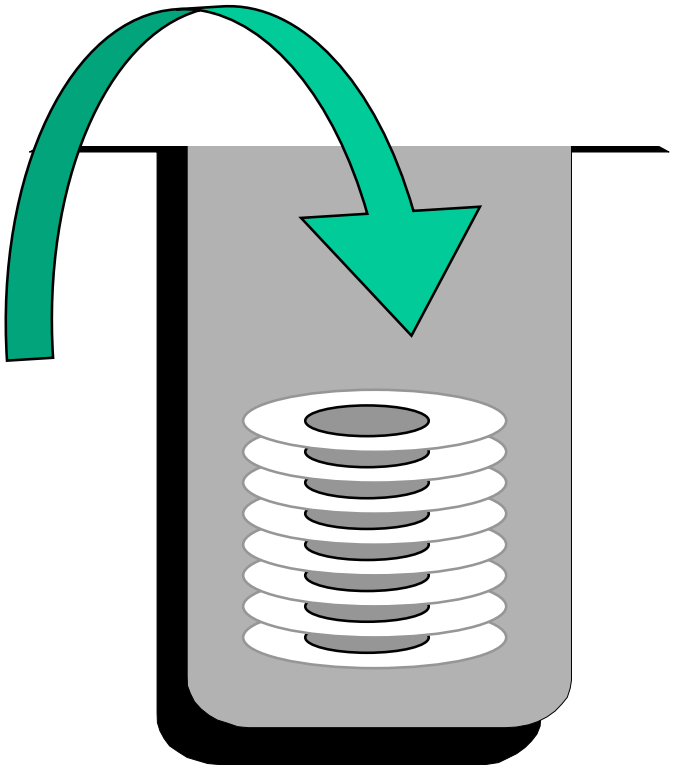
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo *self-service*
- Os pratos são colocados sempre no topo da pilha pelos empregados...



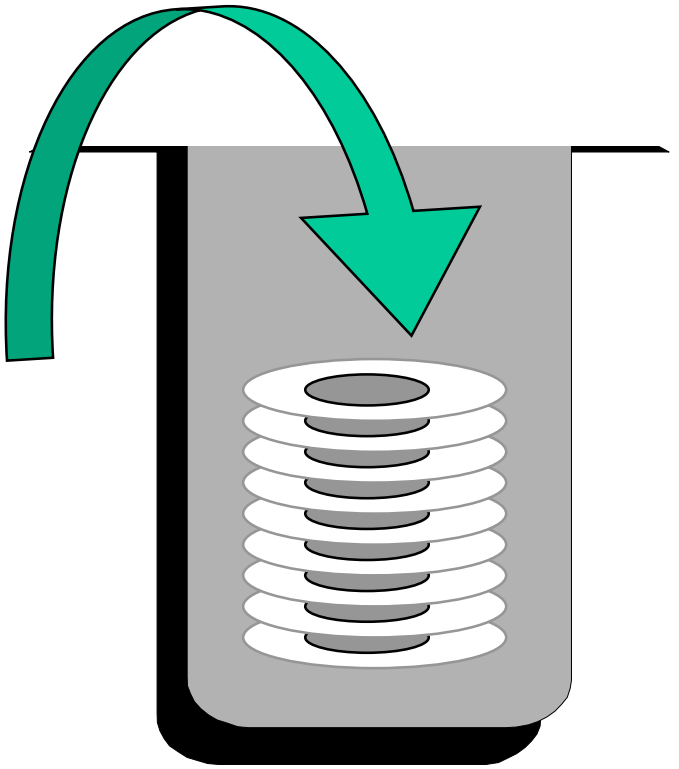
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo *self-service*
- Os pratos são colocados sempre no topo da pilha pelos empregados...



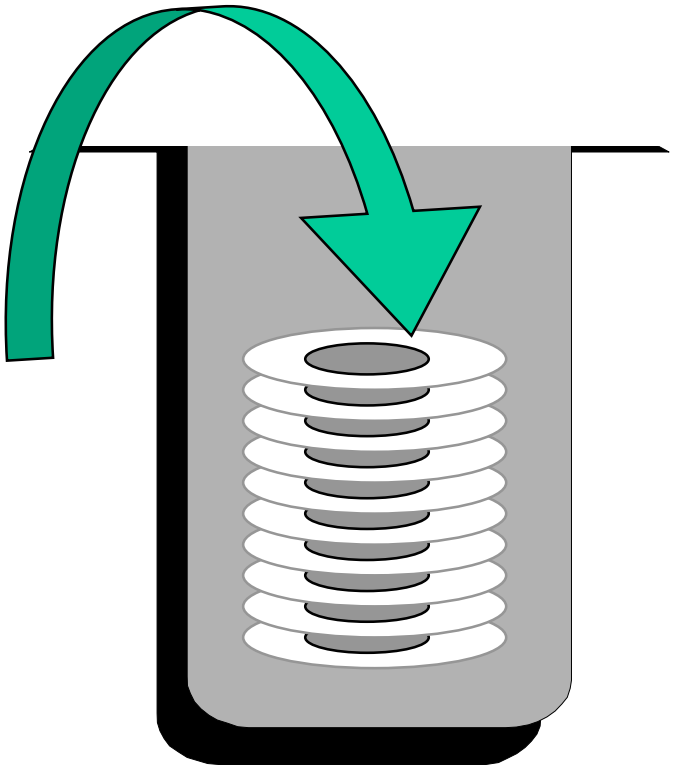
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo *self-service*
- Os pratos são colocados sempre no topo da pilha pelos empregados...



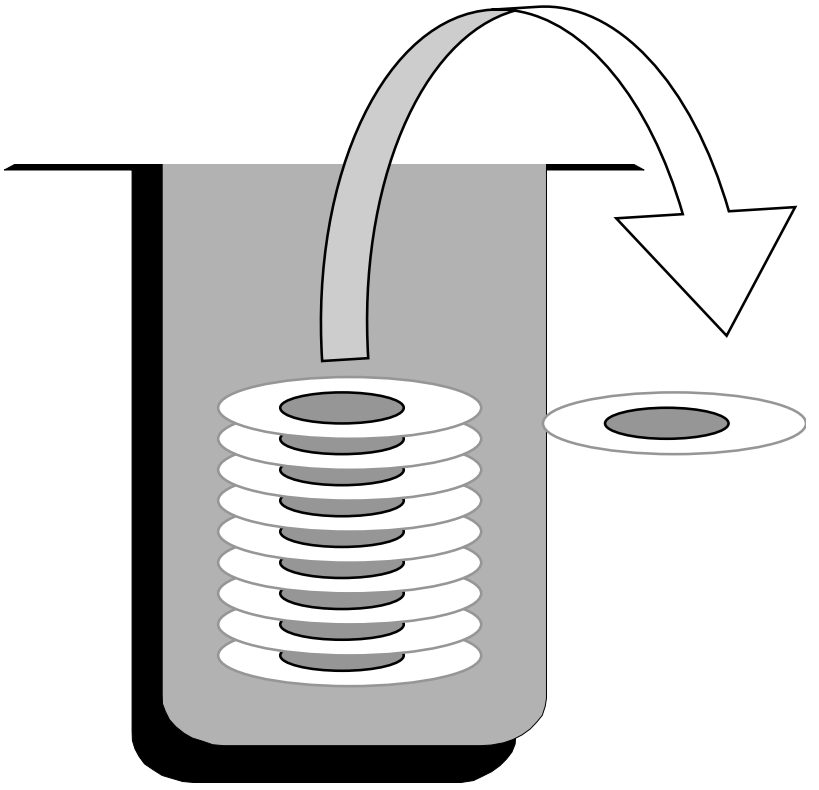
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo *self-service*
- Os pratos são colocados sempre no topo da pilha pelos empregados...



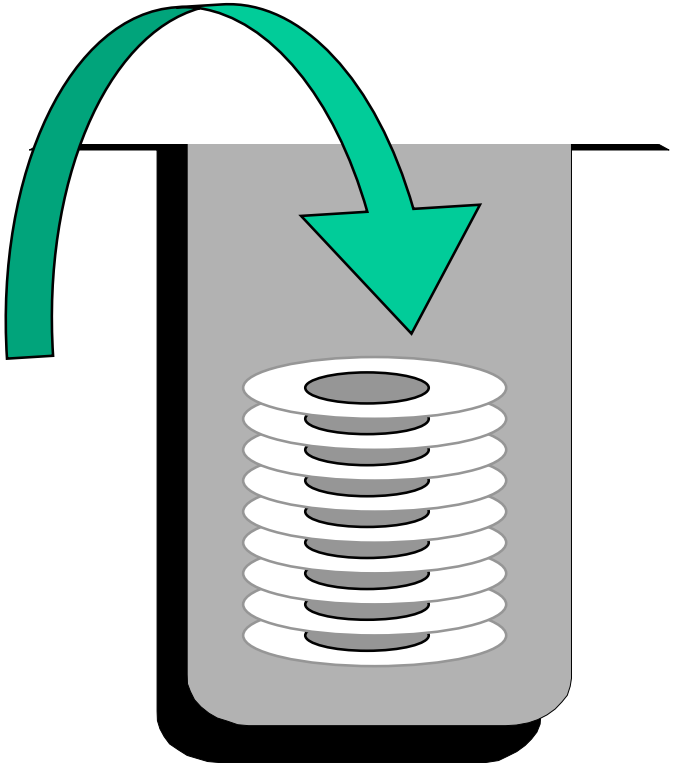
Exemplo

- Por exemplo, pense numa pilha de pratos comumente encontrada em restaurantes do tipo self-service
- Os pratos são colocados sempre no topo da pilha pelos empregados e são retirados a partir do topo pelos clientes
- O prato localizado no fundo da pilha foi o primeiro a ser colocado na pilha e é o último a ser retirado



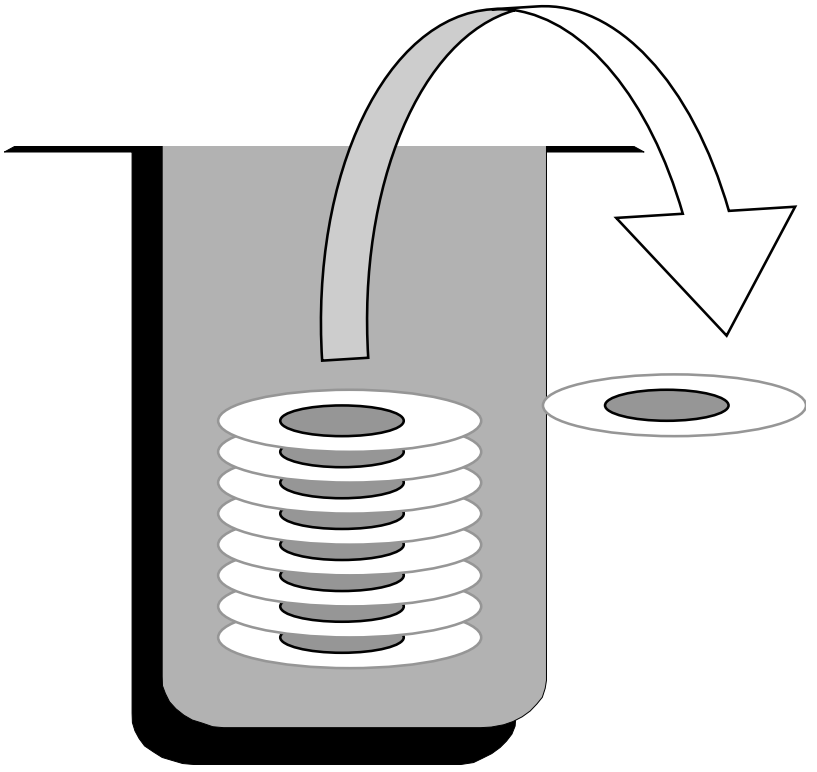
Operações Fundamentais

- Quando um item é adicionado numa pilha, usa-se a operação **Push** (inserir)



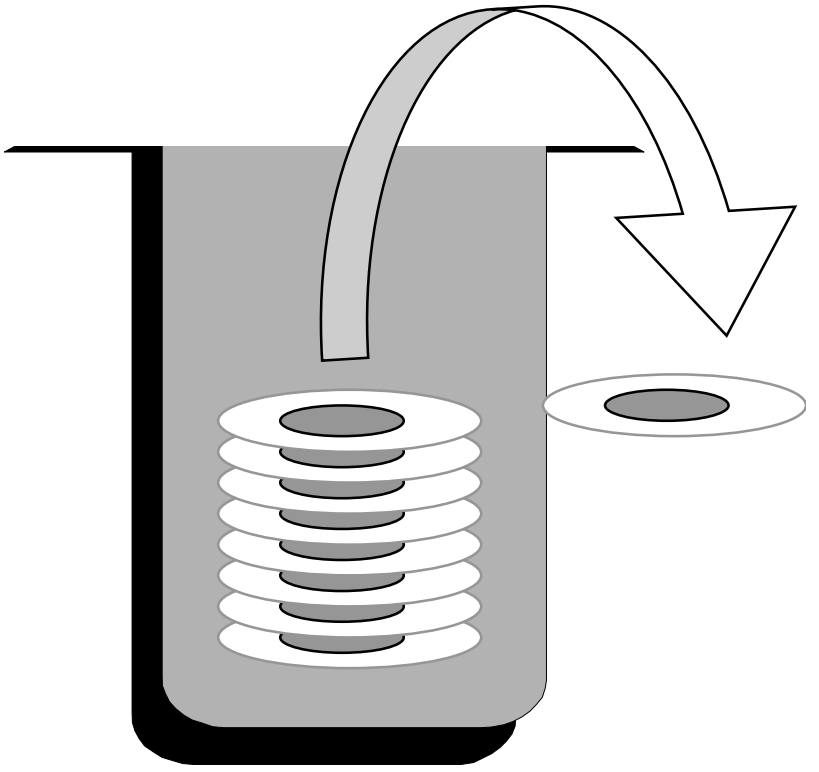
Operações Fundamentais

- Quando um item é adicionado numa pilha, usa-se a operação **Push** (inserir)
- Quando um item é retirado de uma pilha, usa-se a operação **Pop** (retirar)



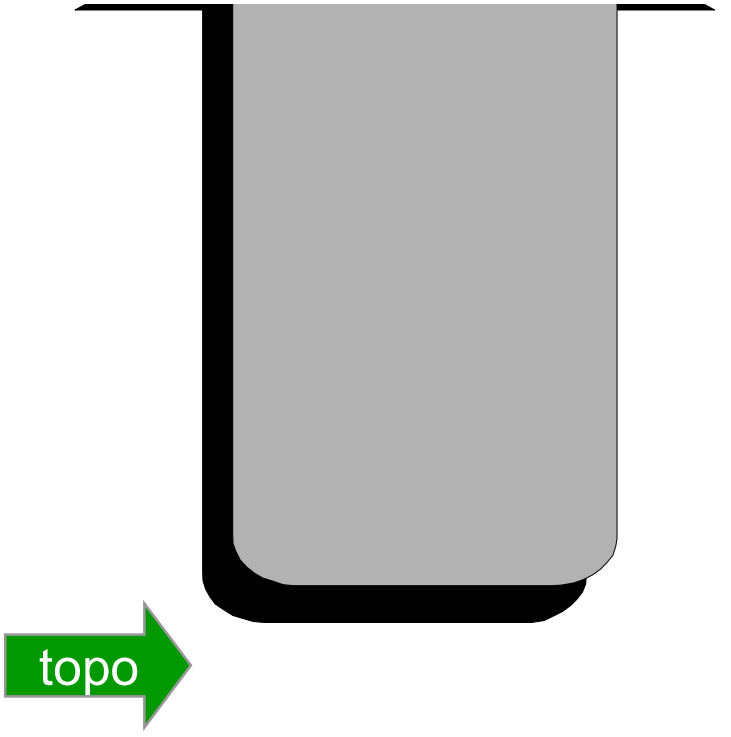
LIFO

- O último item inserido (**Push**) na pilha é sempre o primeiro a ser retirado (**Pop**)
- Esta propriedade é denominada *Last In, First Out* (último a entrar, primeiro a sair) ou **LIFO**



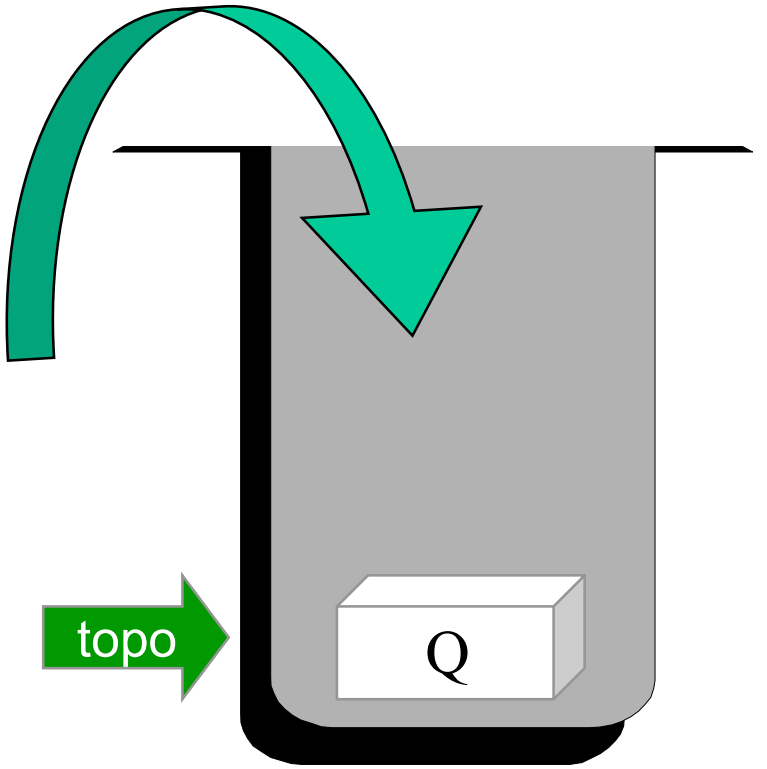
Exemplo

- pilha vazia inicialmente



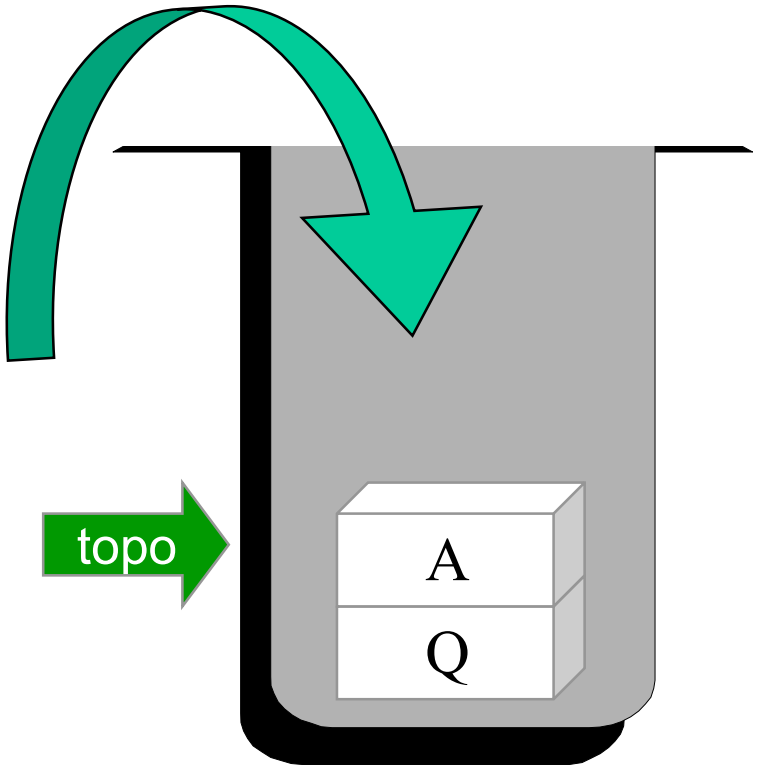
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q



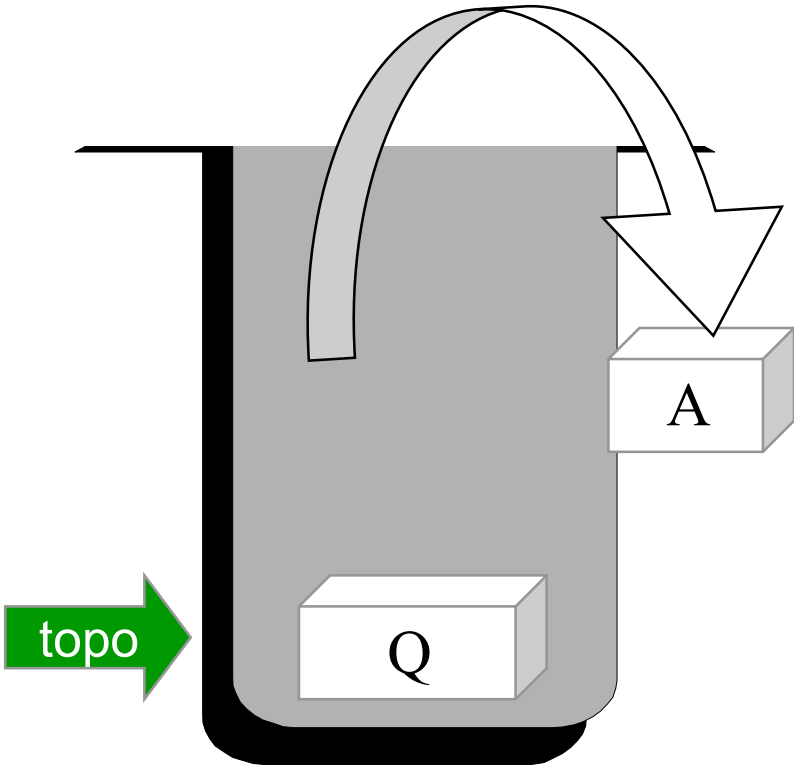
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A



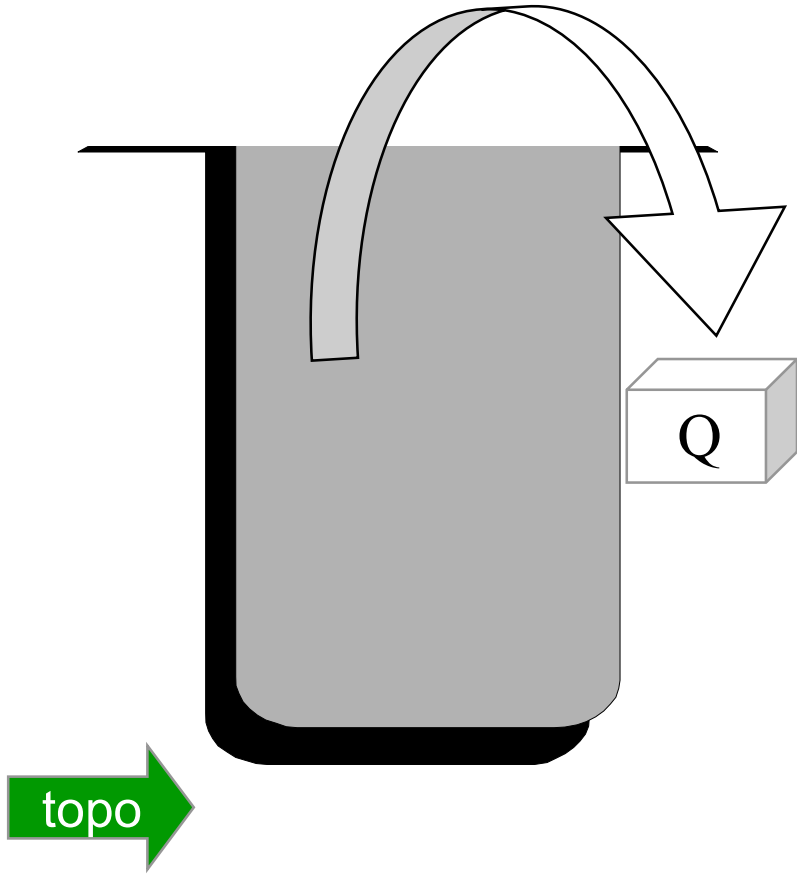
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa



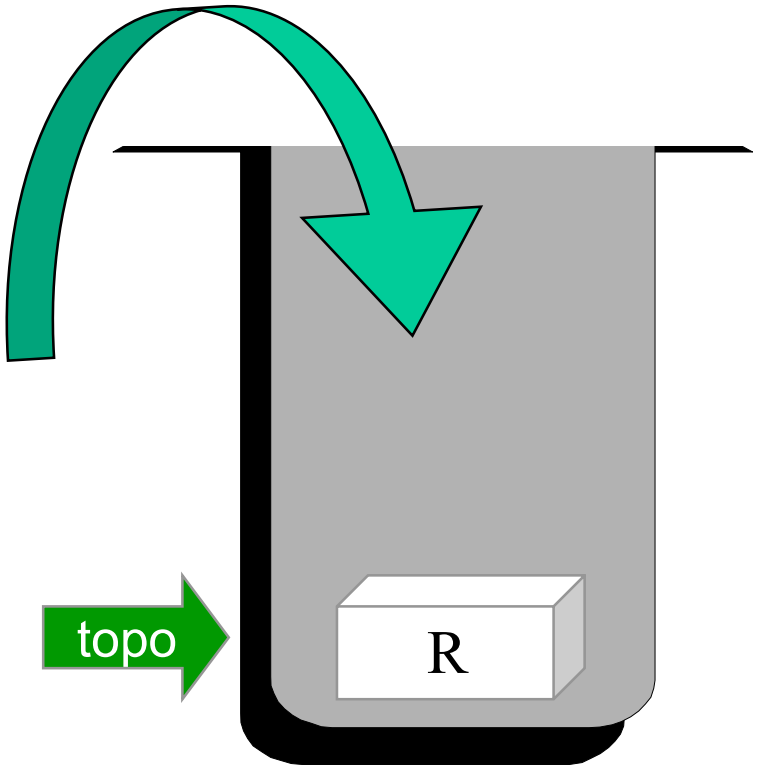
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa
- remover (pop) uma caixa



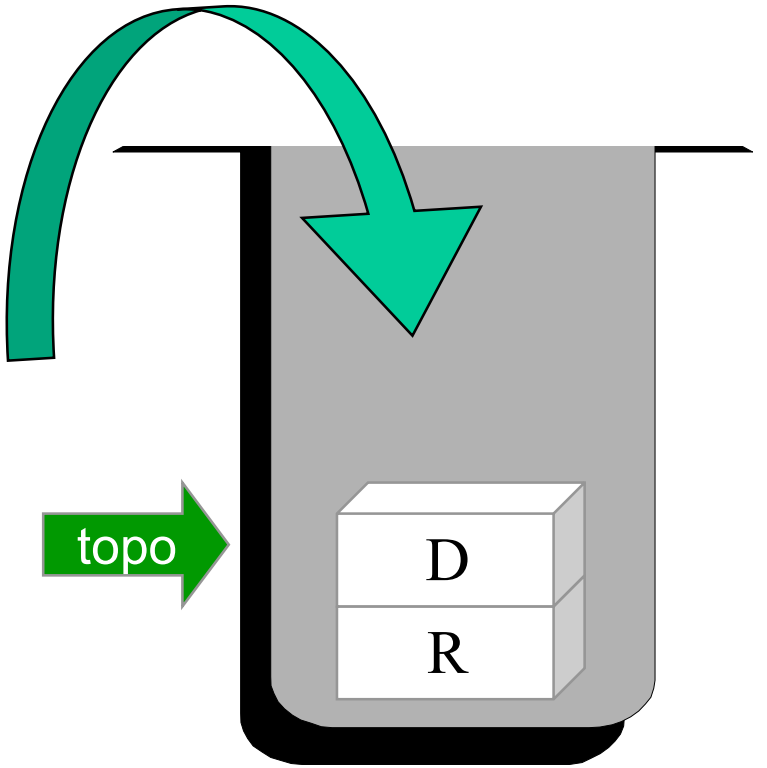
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa
- remover (pop) uma caixa
- inserir (push) caixa R



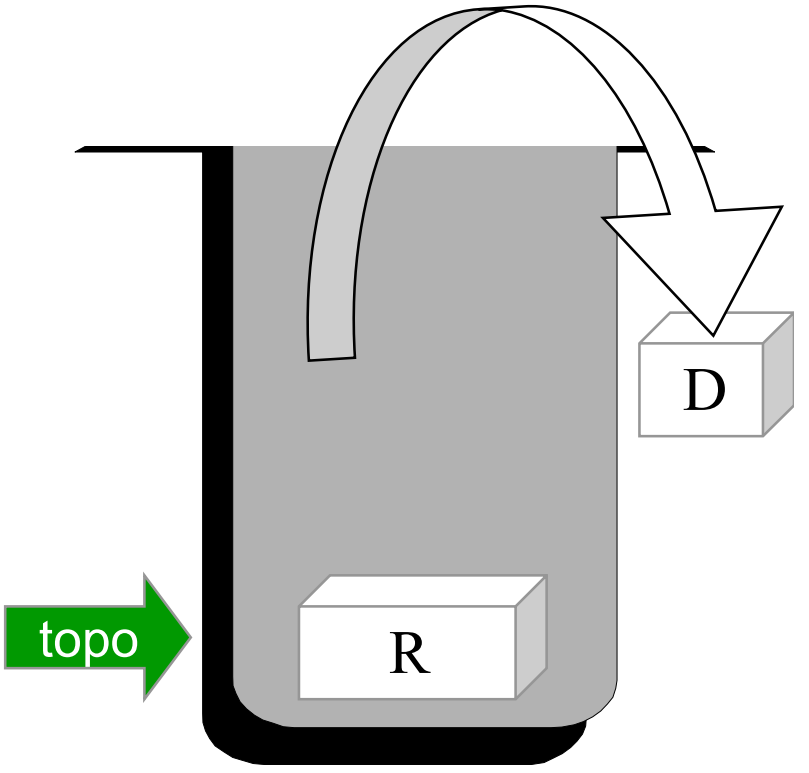
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa
- remover (pop) uma caixa
- inserir (push) caixa R
- inserir (push) caixa D



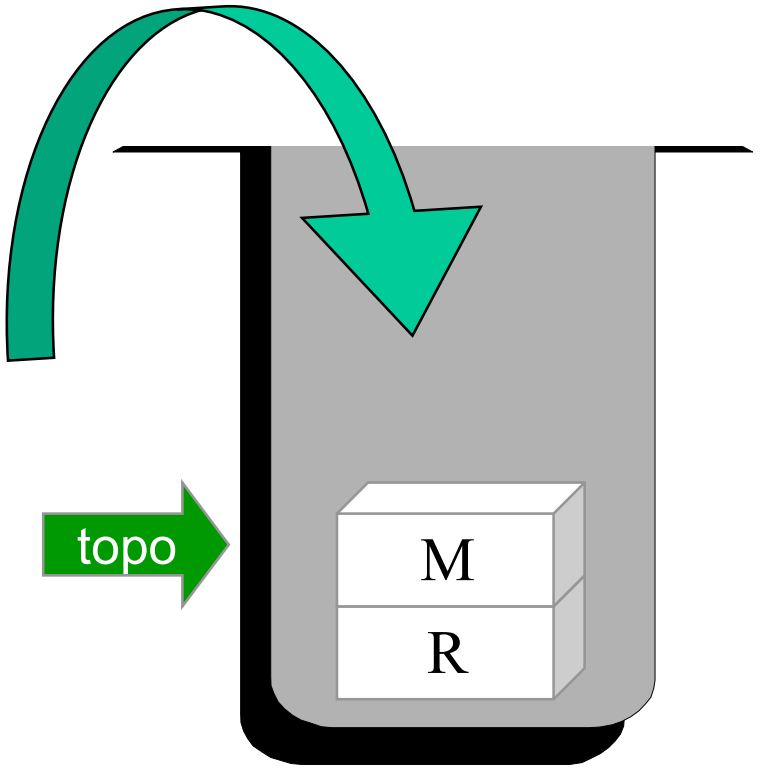
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa
- remover (pop) uma caixa
- inserir (push) caixa R
- inserir (push) caixa D
- remover (pop) uma caixa



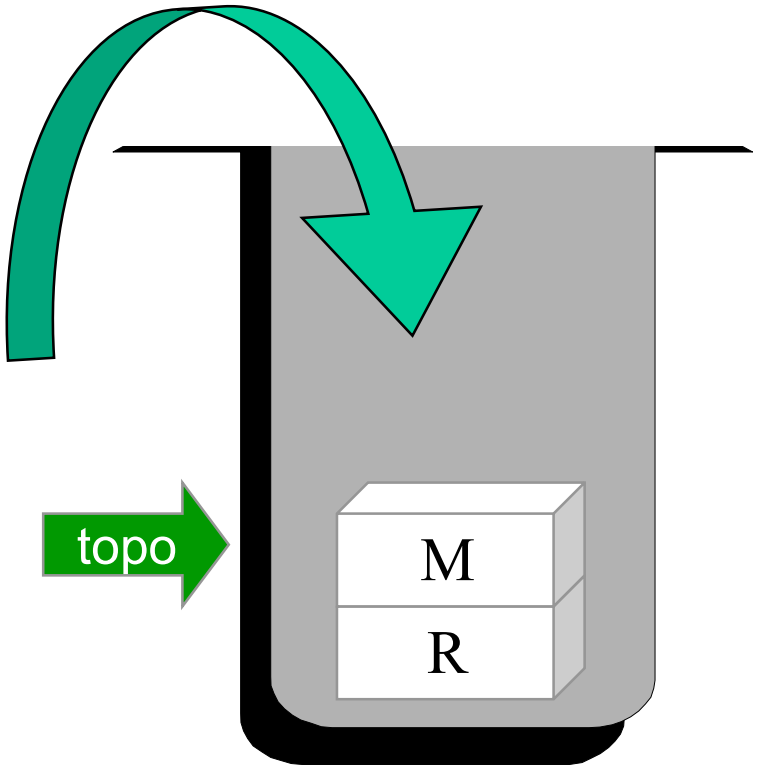
Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa
- remover (pop) uma caixa
- inserir (push) caixa R
- inserir (push) caixa D
- remover (pop) uma caixa
- inserir (push) caixa M



Exemplo

- pilha vazia inicialmente
- inserir (push) caixa Q
- inserir (push) caixa A
- remover (pop) uma caixa
- remover (pop) uma caixa
- inserir (push) caixa R
- inserir (push) caixa D
- remover (pop) uma caixa
- inserir (push) caixa M



Estado final
da pilha

Exemplo: Invertendo uma Linha

- Suponha que você precisa de um procedimento que leia uma linha e escreva-a de forma reversa (de trás para frente)
- Isto pode ser feito simplesmente inserindo (push) cada caracter numa pilha à medida que ele é lido
- Quando a linha estiver terminada, basta retirar (pop) os caracteres da pilha e eles virão em ordem reversa

Invertendo uma Linha

```
void ReverseRead()
// Dada uma linha fornecida pelo usuário, escreve-a em ordem reversa
{ Stack line;
  char ch;

  while (! cin.eof() && ! line.Full())
  {  cin >> ch;                  // Inserir cada caracter da linha para a pilha
     line.Push(ch);
  }
  cout <<  "\nEsta é a linha em ordem reversa:\n";
  while (! line.Empty())
  {  line.Pop(ch);               // Retirar cada caracter da pilha e escrevê-lo
     cout << ch;
  }
  cout << endl;
} // end ReverseRead
```

Information Hiding

- Note que o procedimento **ReverseRead** foi escrito antes de se considerar como realmente a pilha foi implementada
- Isto é um exemplo de **ocultamento de informação** (*information hiding*)
- Se alguém já escreveu os métodos para manipulação de pilhas, então podemos usá-los sem conhecer os detalhes de como as pilhas são mantidas na memória ou como as operações em pilhas são realmente efetuadas

Information Hiding

- *Information hiding* facilita a *mudança de implementação*: se uma implementação é mais eficiente, então apenas as declarações e os métodos de manipulação de pilhas serão alterados
- O programa torna-se mais *legível*: o aparecimento das palavras Push e Pop alertam quem está lendo sobre o que está ocorrendo
- Separando o uso de estruturas de dados da sua implementação ajuda a melhorar o projeto *top-down* (do nível maior de abstração para o menor; do menor detalhamento para o maior) tanto de estruturas de dados quanto de programas

Especificação

- Operações:
  - Criação
  - Destruição
  - Status
  - Operações Básicas
  - Outras Operações

Criação

Stack::Stack();

- *pré-condição:* nenhuma
- *pós-condição:* Pilha é criada e iniciada como vazia

Destruição

Stack::~Stack();

- *pré-condição:* Pilha já tenha sido criada
- *pós-condição:* Pilha é destruída

Status

bool Stack::Empty();

- *pré-condição:* Pilha já tenha sido criada
- *pós-condição:* função retorna **true** se a pilha está vazia; **false** caso contrário

Status

bool Stack::Full();

- *pré-condição:* Pilha já tenha sido criada
- *pós-condição:* função retorna **true** se a pilha está cheia; **false** caso contrário

Operações Básicas

void Stack::Push(StackEntry x);

- *pré-condição:* Pilha já tenha sido criada e não está cheia
- *pós-condição:* O item **x** é armazenado no topo da pilha

O tipo **StackEntry** depende da aplicação e pode variar desde um simples caracter ou número até uma **struct** ou **class** com muitos campos

Operações Básicas

void Stack::Pop(StackEntry &x);

- *pré-condição:* Pilha já tenha sido criada e não está vazia
- *pós-condição:* O item no topo da pilha é removido e seu valor é retornado na variável **x**

Outras Operações

void Stack::Clear();

- *pré-condição:* Pilha já tenha sido criada
- *pós-condição:* todos os itens da pilha são descartados e ela torna-se uma pilha vazia

Outras Operações

int Stack::Size();

- *pré-condição:* Pilha já tenha sido criada
- *pós-condição:* a função retorna o número de itens na pilha

Outras Operações

void Stack::Top(StackEntry &x);

- *pré-condição:* Pilha já tenha sido criada e não está vazia
- *pós-condição:* A variável **x** recebe uma cópia do item no topo da pilha; a pilha permanece inalterada

Pontos Importantes

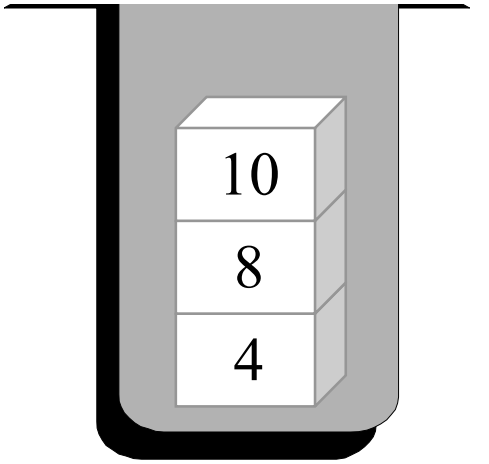
- Uma analogia útil para uma pilha consiste em pensar em uma pilha de pratos: os pratos são colocados e retirados sempre a partir do topo da pilha
- As operações básicas são **Push** (insere um elemento na pilha) e **Pop** (retira um elemento da pilha)

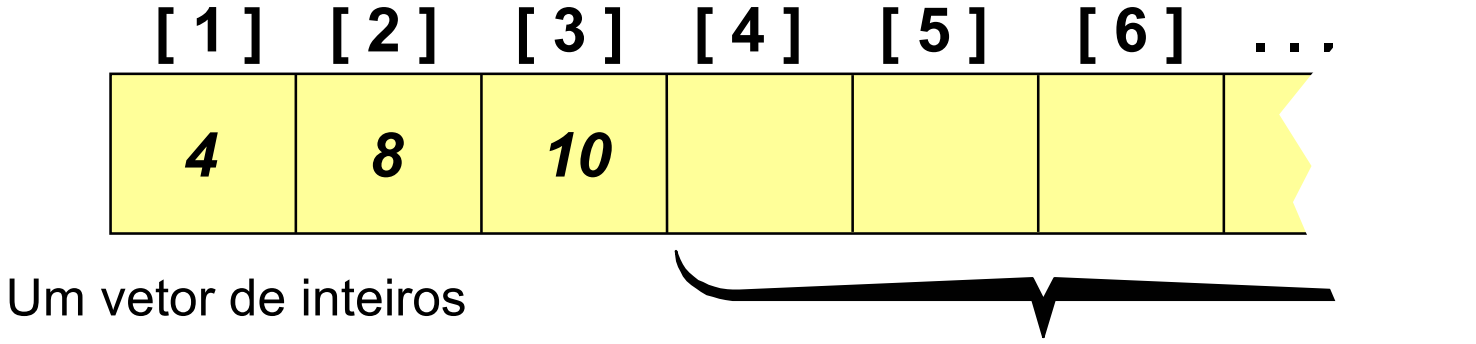
Implementação Contígua

- Veremos inicialmente os detalhes de implementação de pilhas utilizando vetores (*arrays*)
- Este tipo de implementação é também denominada estática e contígua (ou seqüencial)
- Logo a seguir, veremos a implementação encadeada dinâmica de pilhas

Implementação Contígua

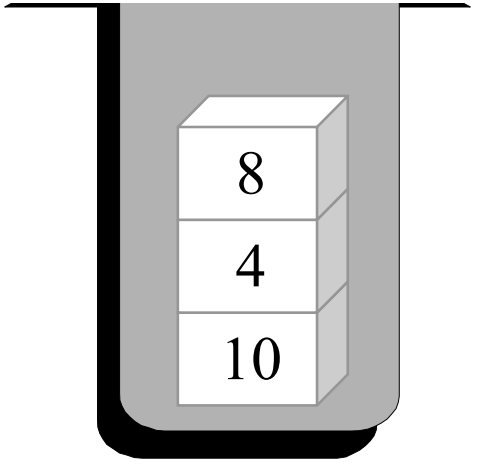
- As entradas em uma pilha serão armazenadas no **início de um vetor**, como mostra este exemplo

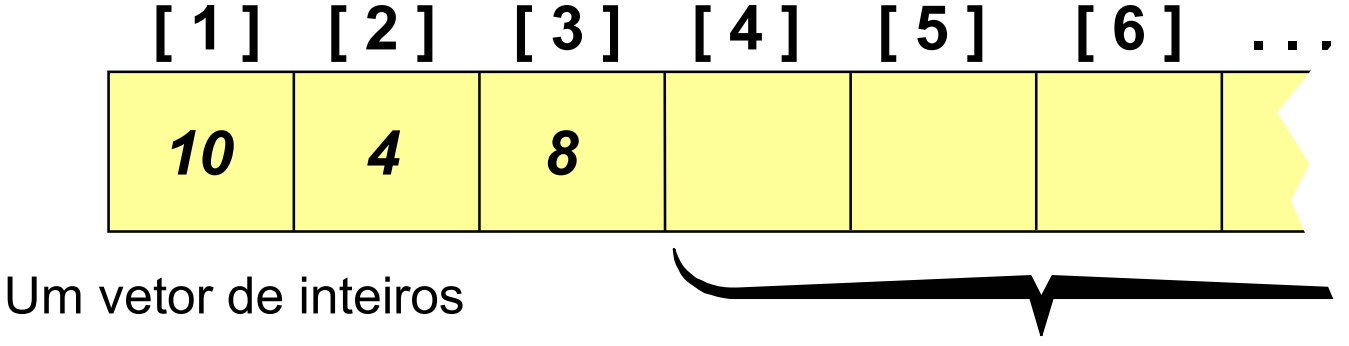




Implementação Contígua

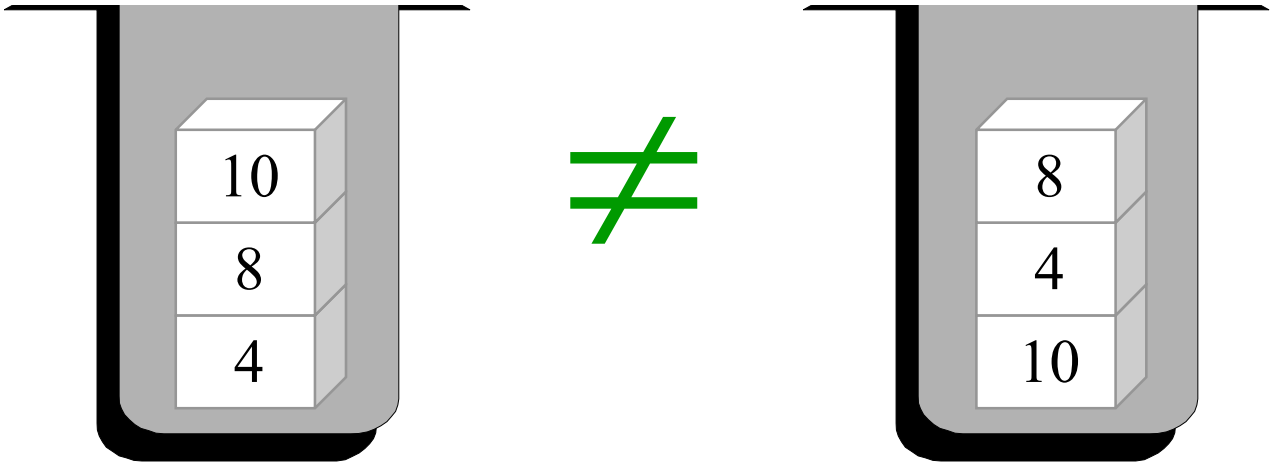
- As entradas possuem ordem. A figura abaixo **não** representa a mesma pilha que a anterior...





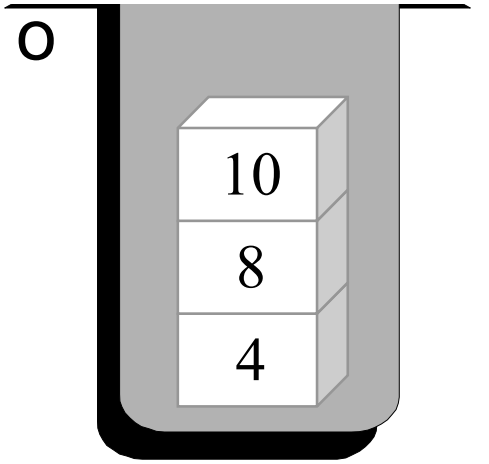
Implementação Contígua

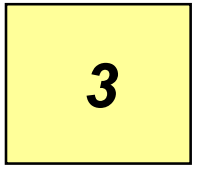
- Assim...



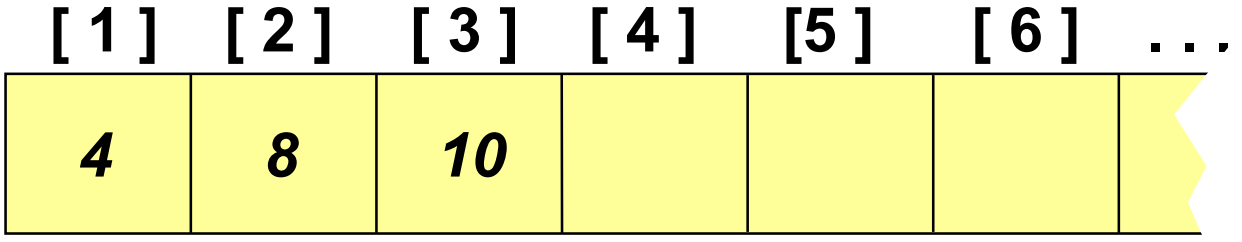
Implementação Contígua

- Nós precisamos também armazenar o topo da pilha





Um inteiro armazena o topo da pilha

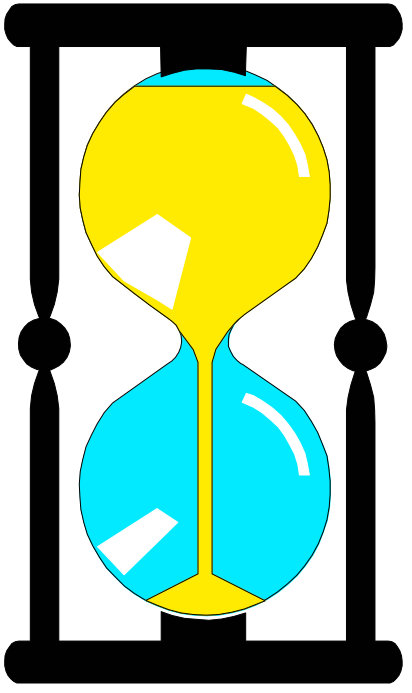


Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor

Questão

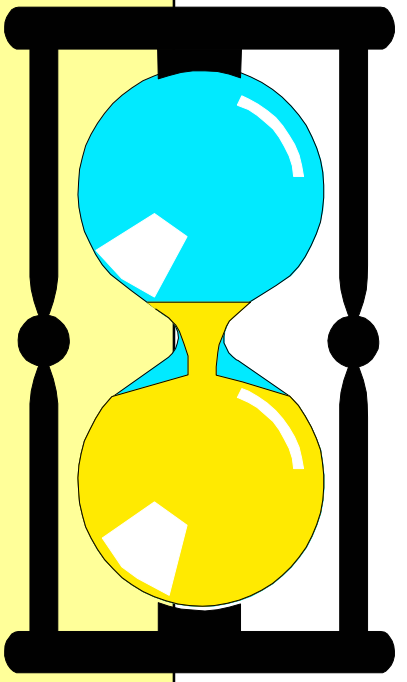
Utilize estas idéias para escrever uma declaração de tipo que poderia implementar o tipo de dado pilha. A declaração deve ser um objeto com dois campos de dados. Faça uma pilha capaz de armazenar 100 inteiros



Você tem 3 minutos para escrever a declaração

Uma Solução

```
const int MaxStack = 100;
class Stack
{ public:
     Stack();
     void Push(int x);
     void Pop(int &x);
     ...
  private:
    int top;                          // topo da pilha
    int Entry[MaxStack+1];  // vetor com elementos
};
```



Uma Solução

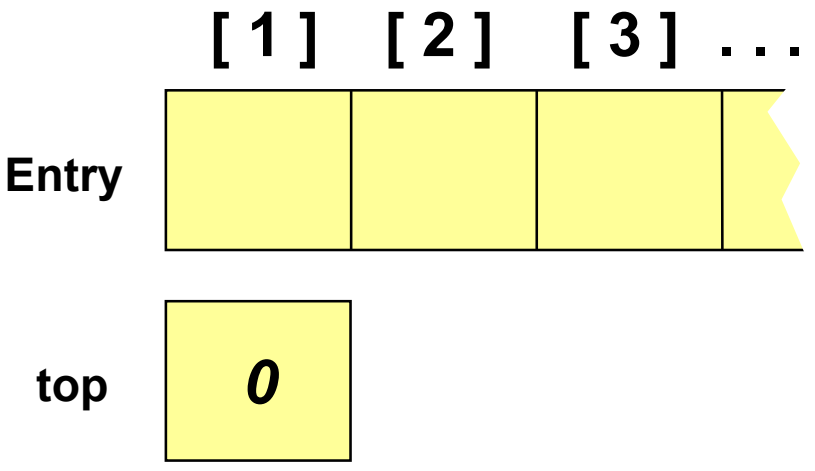
```
const int MaxStack = 100;
class Stack
{ public:
     Stack();
     void Push(int x);
     void Pop(int &x);
     ...
  private:
    int top;                          // topo da pilha
    int Entry[MaxStack+1];  // vetor com elementos
};
```

Observe que o tipo **StackEntry** nesse caso é um inteiro

Criação

Stack::Stack()

A pilha deve iniciar vazia. A convenção é que **top** = 0

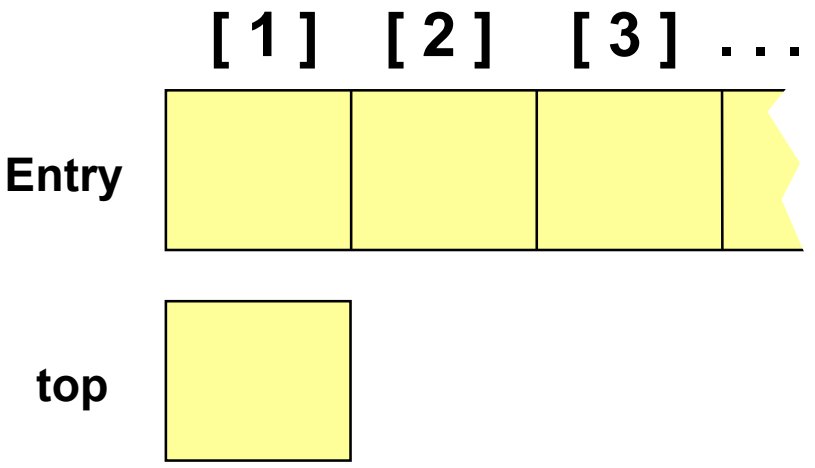



```
Stack::Stack()
{
    top = 0;
}
```

Destruidor

Stack::~Stack()

Usando alocação estática para implementar a pilha (vetor), o destruidor não será necessário. Em todo caso, colocaremos apenas uma mensagem que o objeto foi destruído

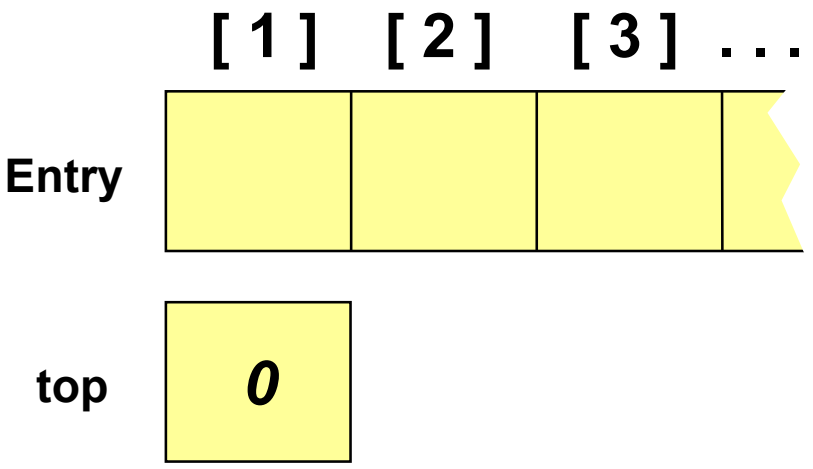



```
Stack::~Stack()
{
    cout << “Pilha destruída”;
}
```

Status: Empty

bool Stack::Empty()

Lembre-se que a pilha inicia vazia, com **top** = 0...

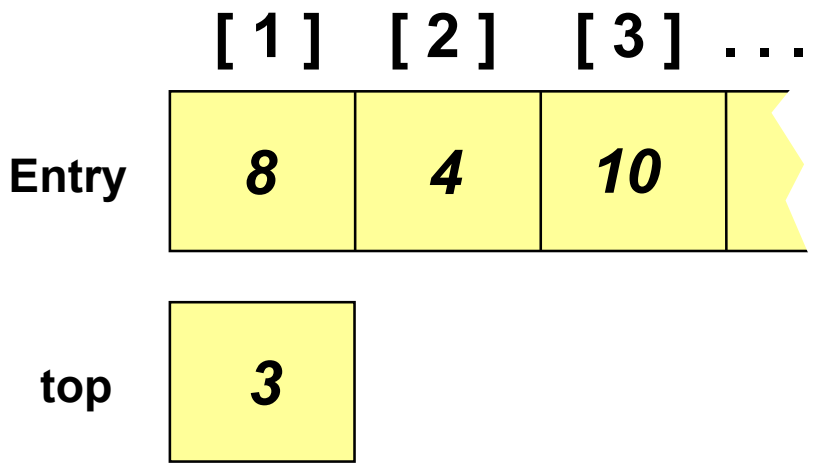



```
bool Stack::Empty()
{
    return (top == 0);
}
```

Status: Full

bool Stack::Full()

...e que **MaxStack** é o número máximo de elementos da pilha

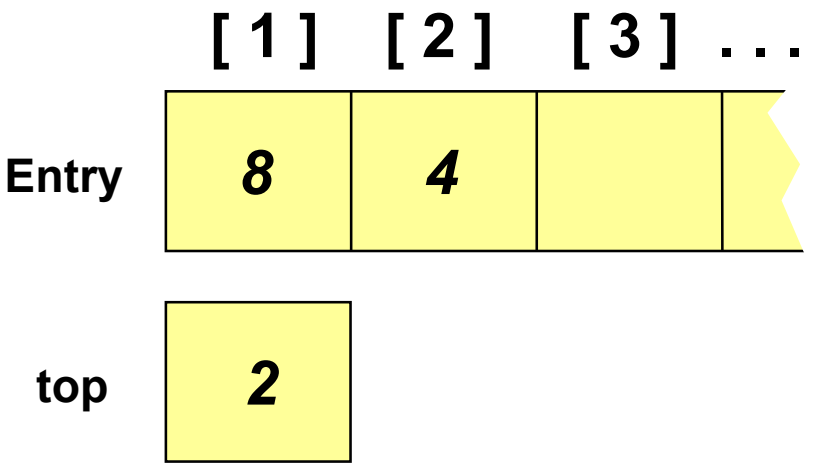



```
bool Stack::Full()
{
   return (top == MaxStack);
}
```

Operações Básicas: Push

```
void Stack::Push(int x)
```

Nós fazemos uma chamada a
Push(17)

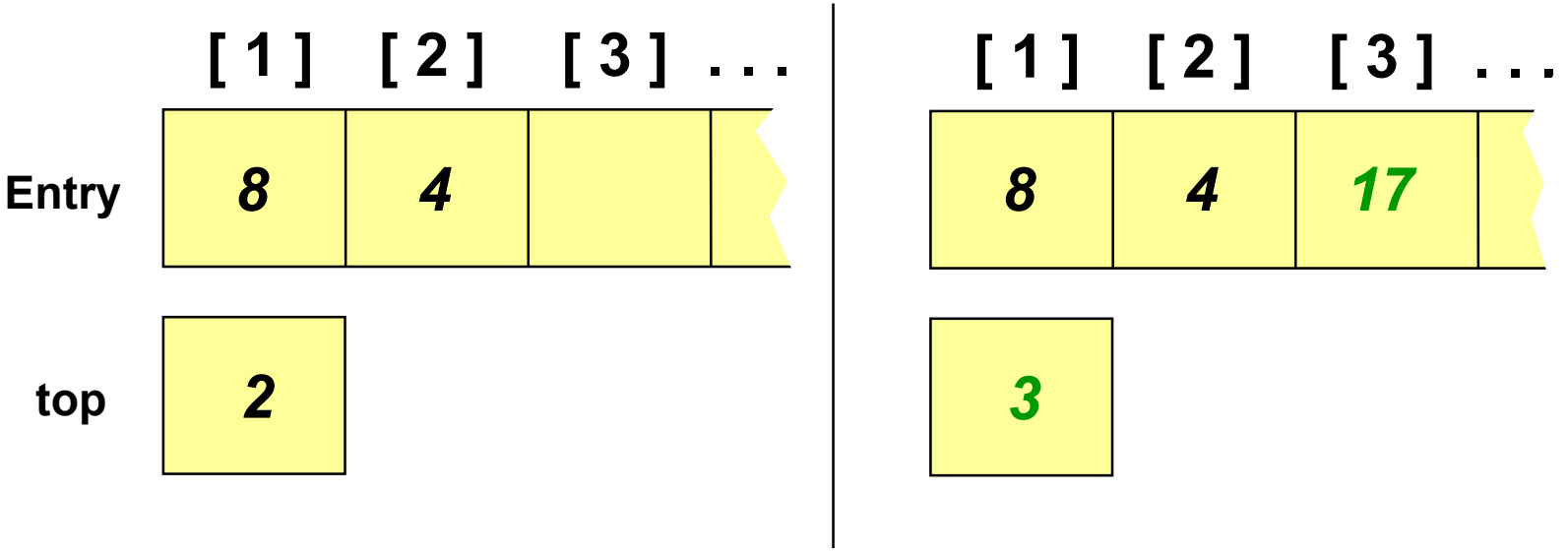


Quais valores serão armazenados em **Entry** e **top** depois que a chamada de procedimento termina?

Operações Básicas: Push

```
void Stack::Push(int x)
```

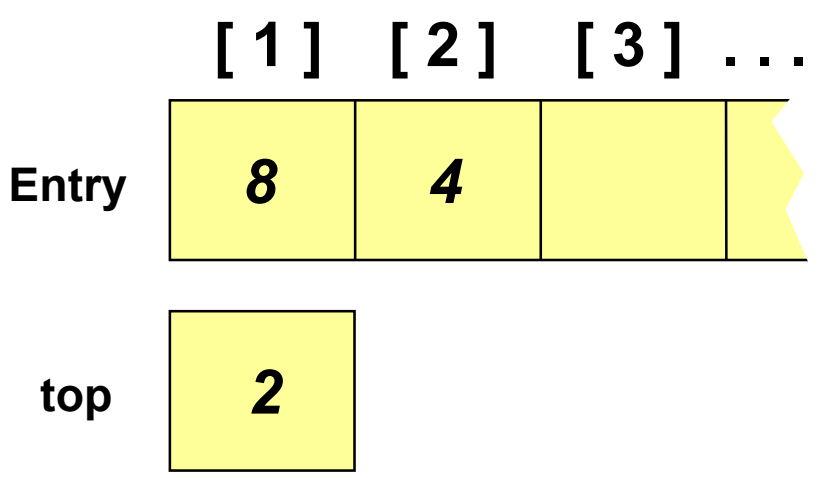
Depois da chamada a Push(17), nós teremos esta pilha:



Operações Básicas: Push

void Stack::Push(int x)

Antes de inserir, é conveniente verificar se há espaço na pilha

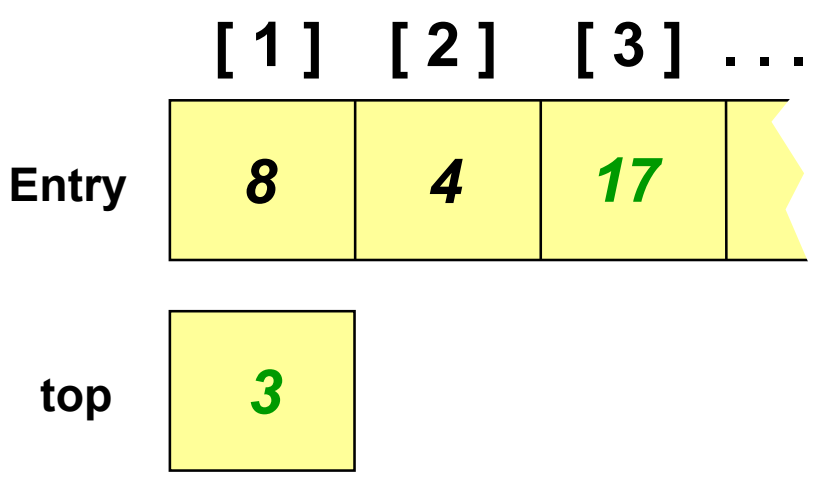



```
void Stack::Push(int x)
{  if (Full())
   {  cout << “Pilha Cheia”;
      abort();
   }
   ...


}
```

Operações Básicas: Push

void Stack::Push(int x)

Se houver, basta inserir na próxima posição livre do vetor

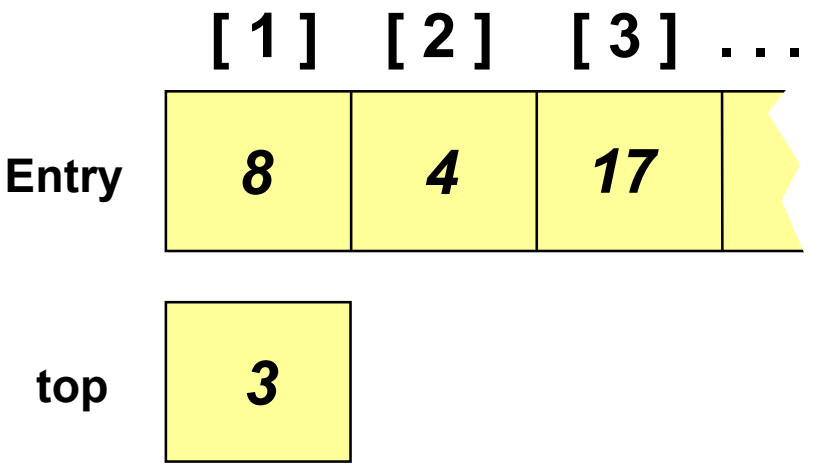



```
void Stack::Push(int x)
{  if (Full())
   {  cout << “Pilha Cheia”;
      abort();
   }

   top++;
   Entry[top] = x;
}
```

Operações Básicas: Pop

```
void Stack::Pop(int &x)
```

Nós fazemos uma chamada a
Pop(x)

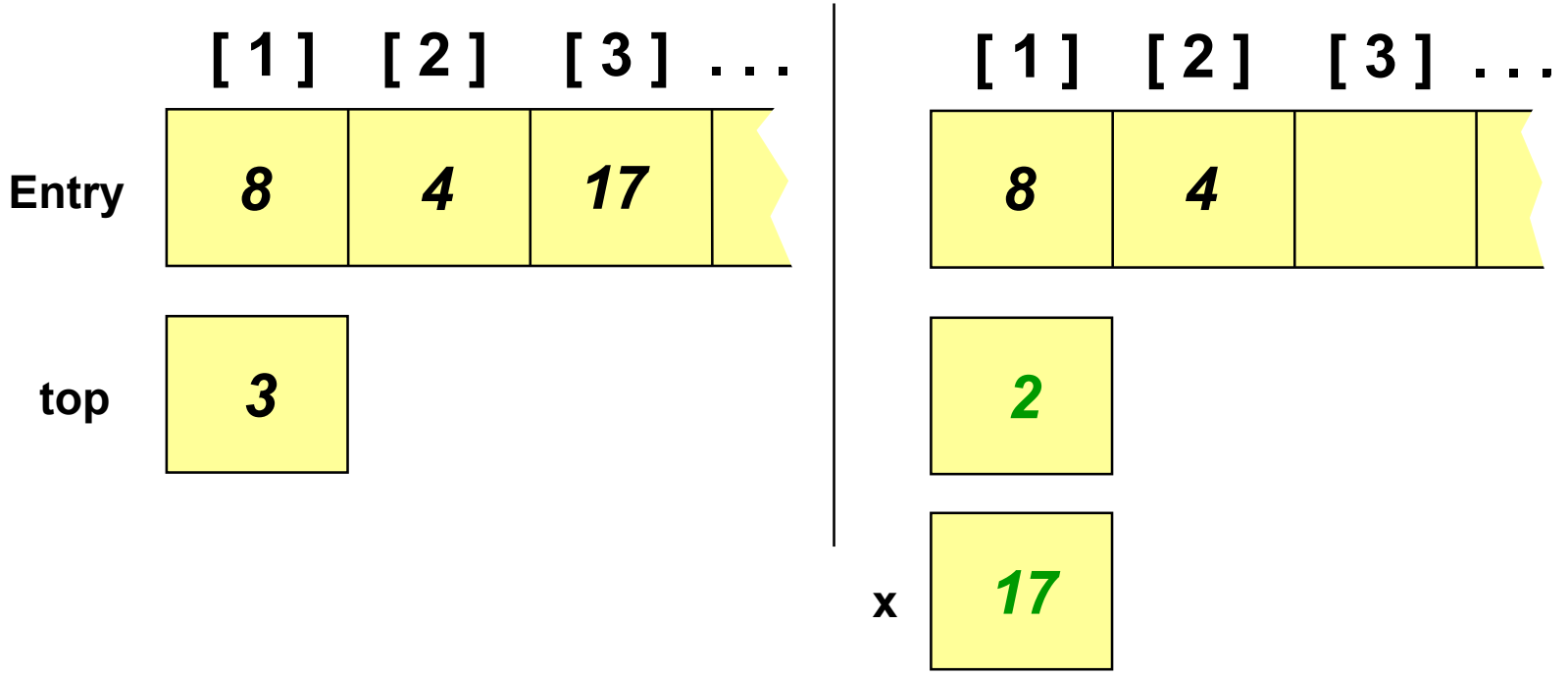


Quais valores serão armazenados em **Entry**, **top** e **x** depois que a chamada de procedimento termina?

Operações Básicas: Pop

```
void Stack::Pop(int &x)
```

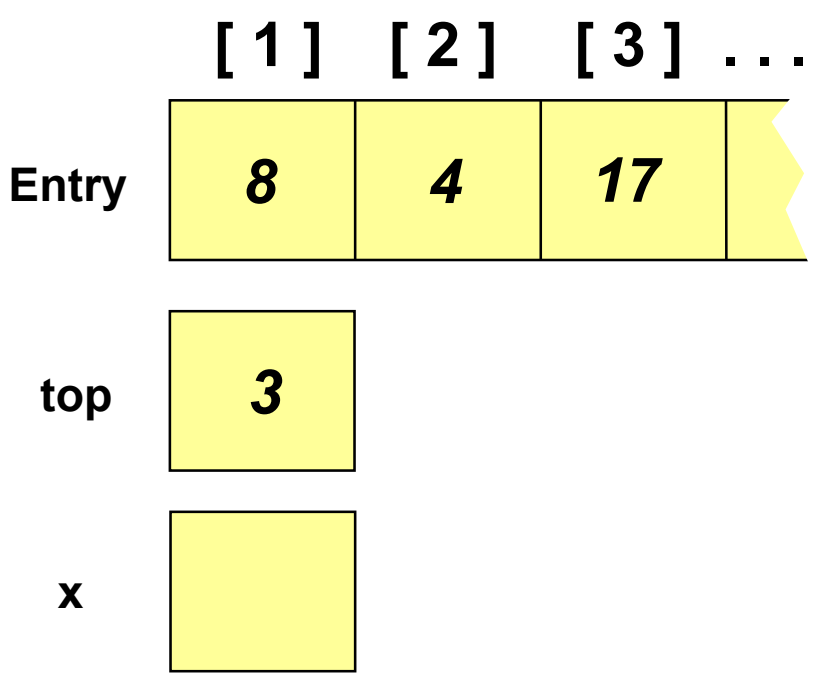
Depois da chamada a Pop(x), nós teremos esta pilha:



Operações Básicas: Pop

void Stack::Pop(int &x)

Antes de remover, é conveniente verificar se a pilha não está vazia

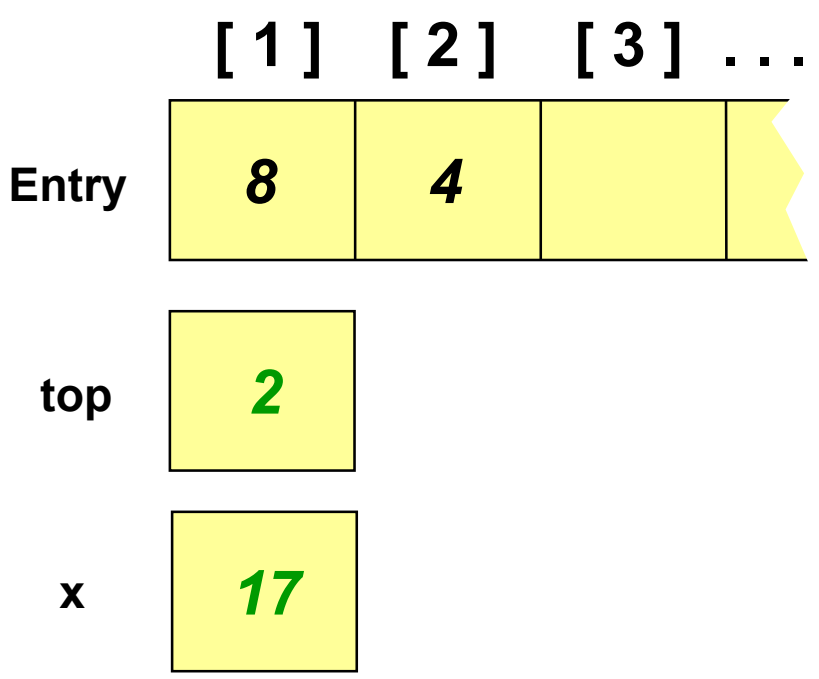



```
void Stack::Pop(int &x)
{  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }
  ...



}
```

Operações Básicas: Pop

void Stack::Pop(int &x)

Se não estiver vazia, basta retirar o elemento do topo do vetor




```
void Stack::Pop(int &x)
{  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }

   x = Entry[top] ;
   top = top - 1;
}
```

Exercícios

- Defina a operação Clear utilizando apenas as operações Empty e Pop
- Defina a operação Clear utilizando diretamente com os campos do objeto (vetor e topo)
- Defina a operação Top utilizando apenas Push e Pop e Empty
- Defina a operação Top utilizando diretamente com os campos do objeto (vetor e topo)
- Defina a operação Size
- Quais as vantagens e desvantagens de cada uma destas construções?

Solução Clear

- Usando apenas Empty e Pop

```
void Stack::Clear()
{ int x;

  while(! Empty())
    Pop(x);
}
```

- Utilizando campos do objeto

```
void Stack::Clear()
{
   top = 0;
}
```

Solução Top

- Usando apenas Push e Pop

```
void Stack::Top(int &x)
{  if(Empty())
   { cout << “Pilha vazia”;
     abort();
   }
   Pop(x);
   Push(x);
}
```

- Utilizando campos do objeto

```
void Stack::Top(int &x)
{  if(top == 0)
   { cout << “Pilha vazia”;
     abort();
   }
   x = Entry[top];
}
```

Solução Size

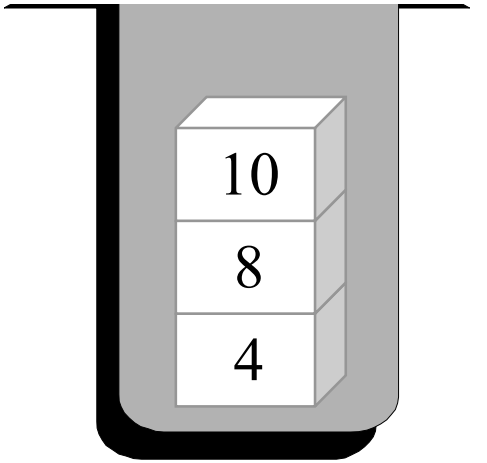
```
int Stack::Size()
{
   return top;
}
```

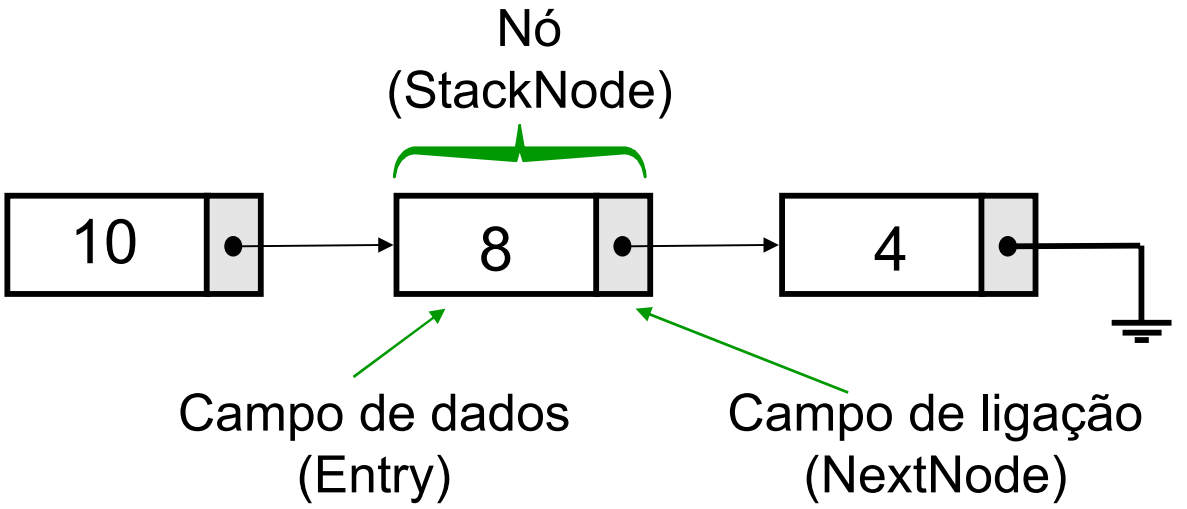
Pontos Importantes

- Até este ponto vimos uma forma de implementação de pilhas, usando vetores
- A vantagem de usar vetores é a simplicidade dos programas
- Uma desvantagem deste tipo de implementação é que o tamanho da pilha é definido *a priori* pelo programador, desperdiçando espaço não utilizado
- Nos próximos *slides*, veremos uma implementação diferente, que otimiza a quantidade de memória utilizada

Implementação Dinâmica

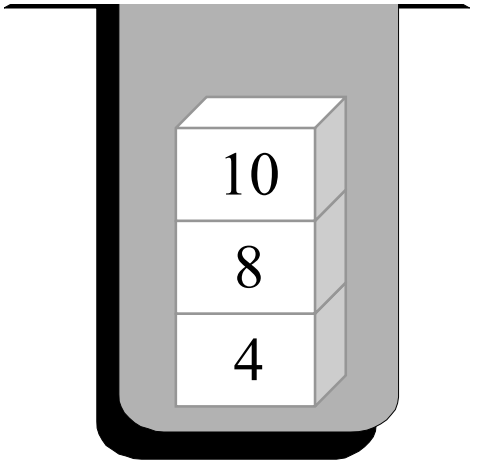
- As entradas são colocadas em uma estrutura (**StackNode**) que contém um campo com o valor existente na pilha (**Entry**) e outro campo é um apontador para o próximo elemento na pilha (**NextNode**)



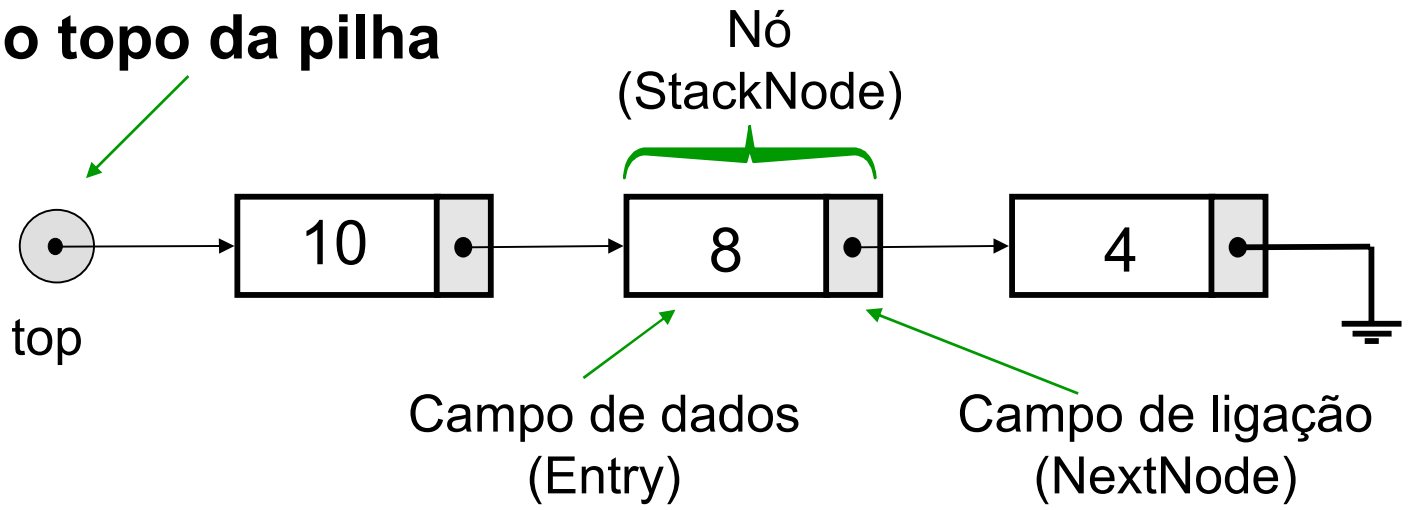


Implementação Dinâmica

- Nós precisamos também armazenar o topo da pilha

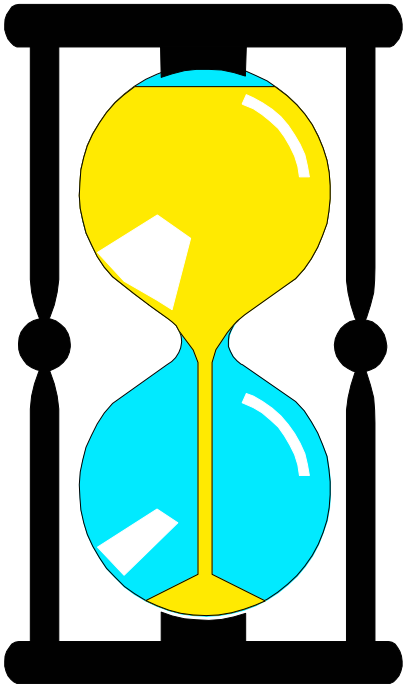


Um ponteiro armazena o topo da pilha



Questão

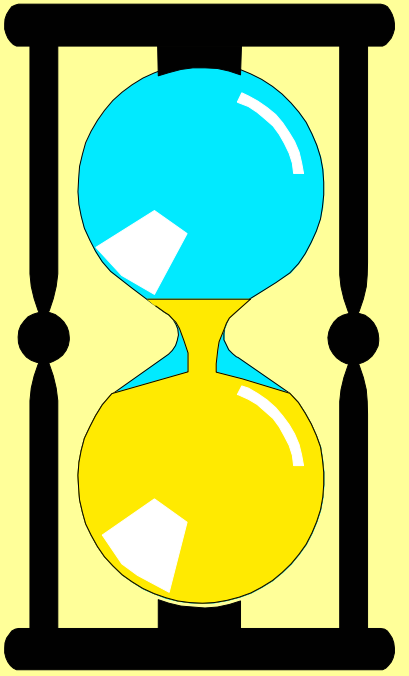
Utilize estas idéias para escrever uma declaração de tipo que poderia implementar o tipo de dado pilha capaz de armazenar inteiros. A declaração deve ser um objeto com dois campos de dados



Você tem 5 minutos para escrever a declaração

Uma Solução

```
class Stack
{ public:
     Stack();
     void Push(int x);
     void Pop(int &x);
     ...
 private:
    // declaração de tipos
    struct StackNode
    { int Entry;       // tipo de dado colocado na pilha
      StackNode *NextNode;   // ligação para próximo
                                        //  elemento na pilha
    };
    typedef StackNode *StackPointer;

    // declaração de campos
    StackPointer top;  // Topo da pilha
};
```



Uma Solução

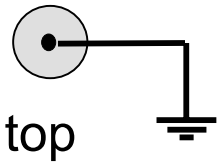
```
class Stack
{ public:
     Stack();
     void Push(int x);
     void Pop(int &x);
     ...
 private:
    // declaração de tipos
    struct StackNode
    { int Entry;       // tipo de dado colocado na pilha
      StackNode *NextNode;   // ligação para próximo
                             //  elemento na pilha
    };
    typedef StackNode *StackPointer;

    // declaração de campos
    StackPointer top;  // Topo da pilha
};
```

Observe que o tipo **StackEntry** nesse caso é um inteiro

Criação

Stack::Stack()

A pilha deve iniciar vazia, ou seja,
top está “aterrado”

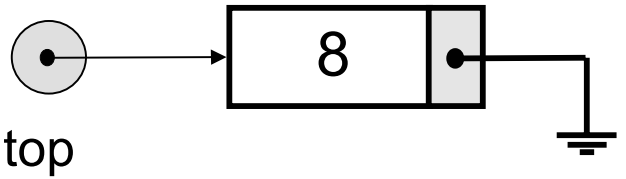



```
Stack::Stack()
{
    top = NULL;
}
```

Destruição

Stack::~Stack()

Usando alocação dinâmica a pilha, o destruidor deve retirar todos os elementos da pilha enquanto ela não estiver vazia. Lembre-se que atribuir NULL a **top** não libera o espaço alocado anteriormente!

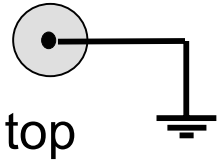



```
Stack::~Stack()
{ int x;
  while ( ! Empty() )
    Pop(x);
}
```

Status: Empty

bool Stack::Empty()

Lembre-se que a pilha inicia vazia, com **top** = NULL...

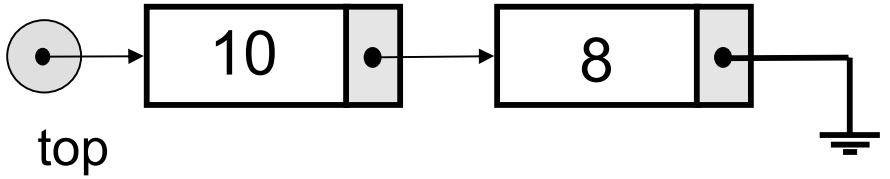



```
bool Stack::Empty()
{
    return (top == NULL);
}
```

Status: Full

bool Stack::Full()

...e que não há limite quanto ao número máximo de elementos da pilha

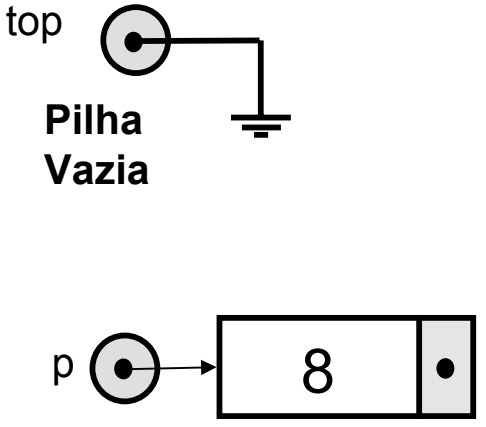


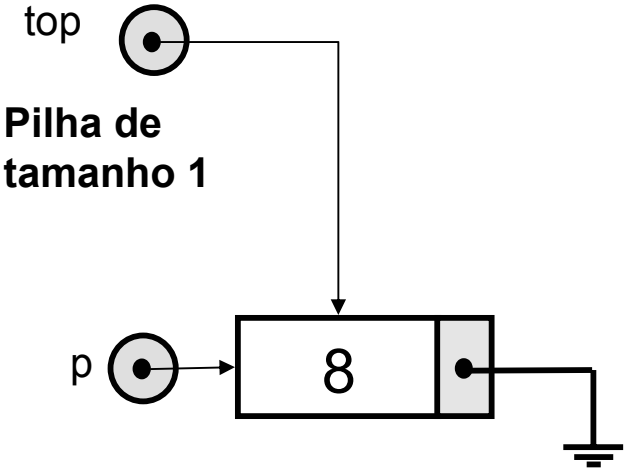

```
bool Stack::Full()
{
   return false;
}
```

Operações Básicas: Push

```
void Stack::Push(int x)
```

Considere agora uma pilha vazia, o que significa **top** = NULL e adicione o primeiro nó. Assuma que esse nó já foi criado em algum lugar na memória e pode ser localizado usando uma variável **p** do tipo ponteiro (StackPointer)





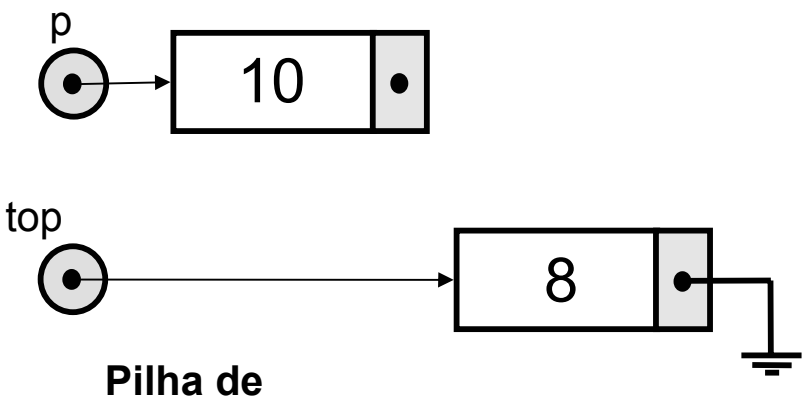
Operações Básicas: Push

void Stack::Push(int x)

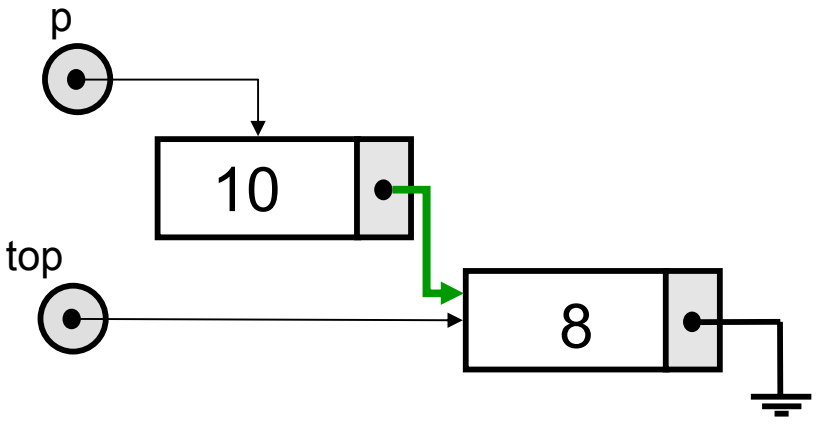
Assim, colocando (Push) **p** na pilha consiste nas instruções:

p->NextNode = top;

...



Pilha de tamanho 1

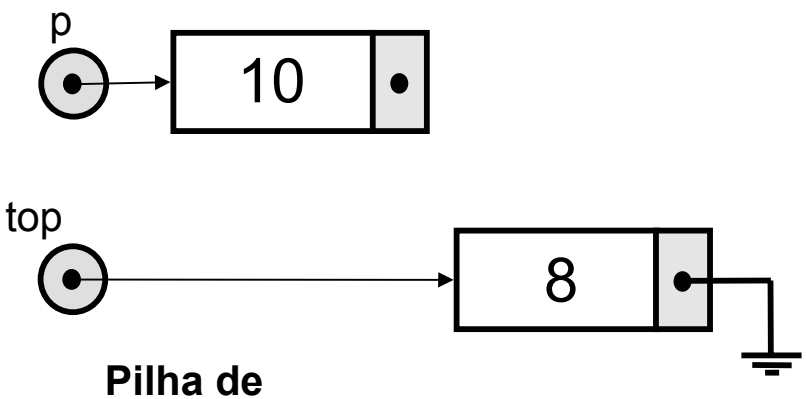


Operações Básicas: Push

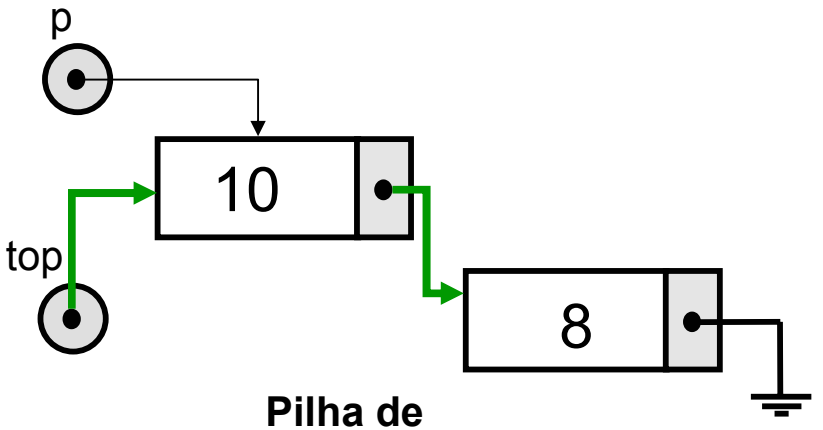
```
void Stack::Push(int x)
```

Assim, colocando (Push) **p** na pilha consiste nas instruções:

```
p->NextNode = top;
top = p;
```



Pilha de tamanho 1



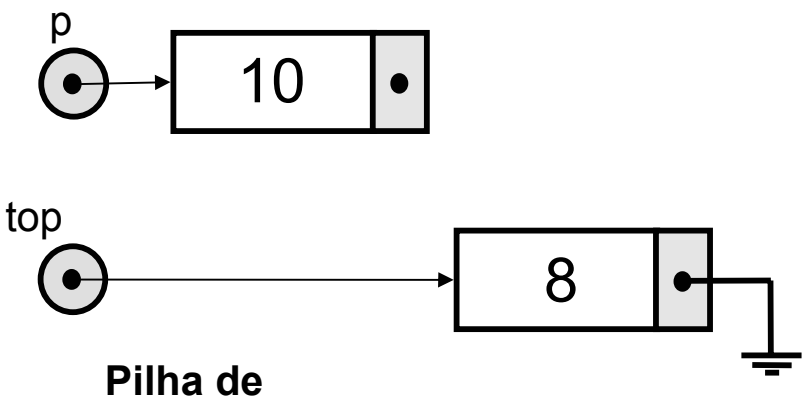
Pilha de tamanho 2

Operações Básicas: Push

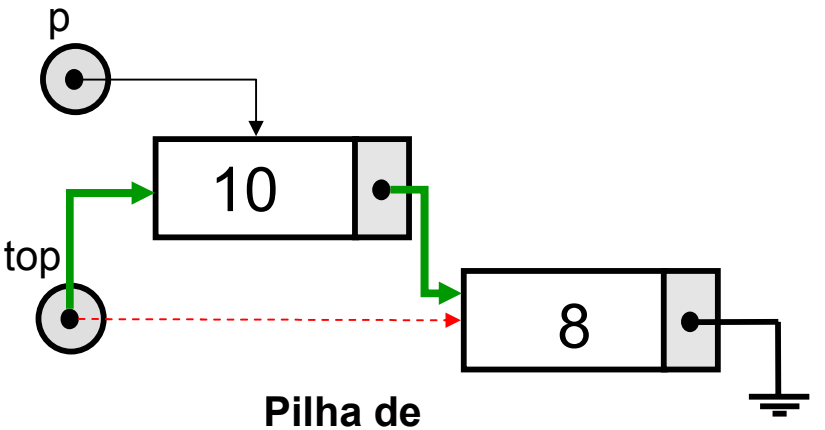
```
void Stack::Push(int x)
```

Assim, colocando (Push) **p** na pilha consiste nas instruções:

```
p->NextNode = top;
top = p;
```



Pilha de tamanho 1



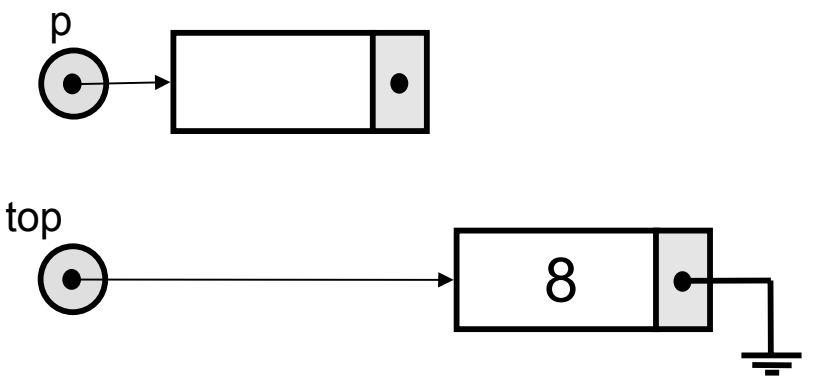
Pilha de tamanho 2

A ligação tracejada foi removida
As ligações em negrito foram adicionadas

Operações Básicas: Push

void Stack::Push(int x)

Inicialmente, alocamos o novo nó, usando o ponteiro **p**

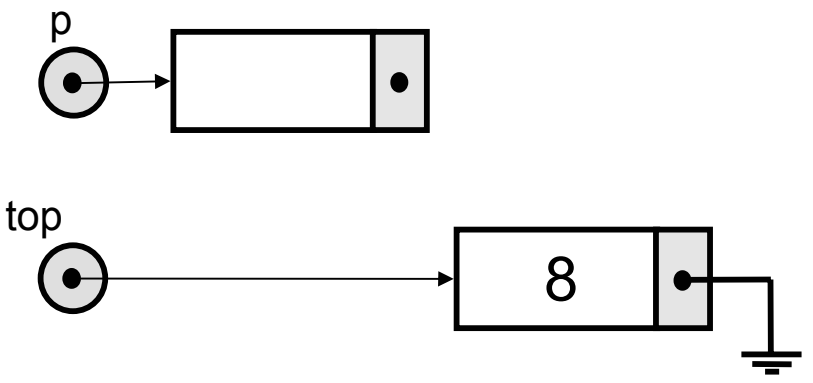



```
void Stack::Push(int x)
{  StackPointer p;

   p = new StackNode;
   ...

}
```

Operações Básicas: Push

void Stack::Push(int x)

Inicialmente, alocamos o novo nó, usando o ponteiro **p**. Se não houver espaço na memória, escrevemos uma mensagem de erro e terminamos

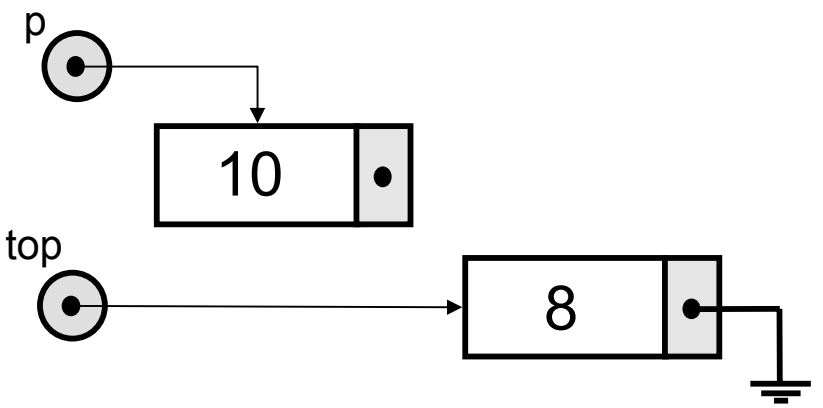



```
void Stack::Push(int x)
{  StackPointer p;

   p = new StackNode;
   if(p == NULL)
   { cout << “Memoria insuficiente”;
     abort();
   }
   ...



}
```

Operações Básicas: Push

void Stack::Push(int x)

Caso contrário, colocamos o elemento **x** no campo de dados de **p**

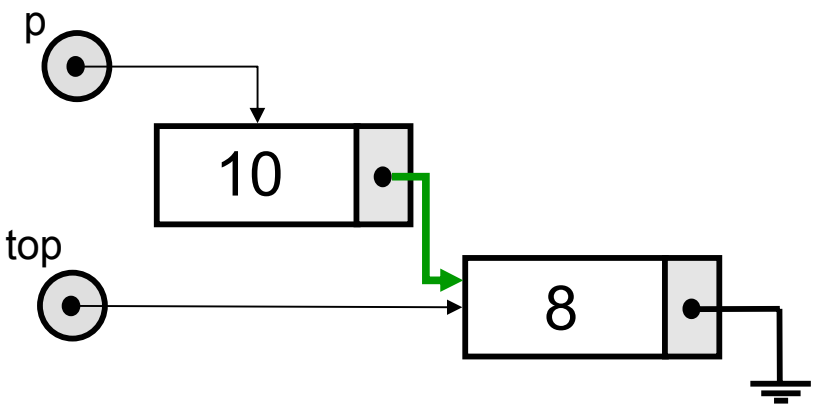



```
void Stack::Push(int x)
{  StackPointer p;

   p = new StackNode;
   if(p == NULL)
   { cout << “Memoria insuficiente”;
     abort();
   }

   p->Entry = x;
   ...

}
```

Operações Básicas: Push

void Stack::Push(int x)

Caso contrário, colocamos o elemento **x** no campo de dados de **p** e alteramos os ponteiros

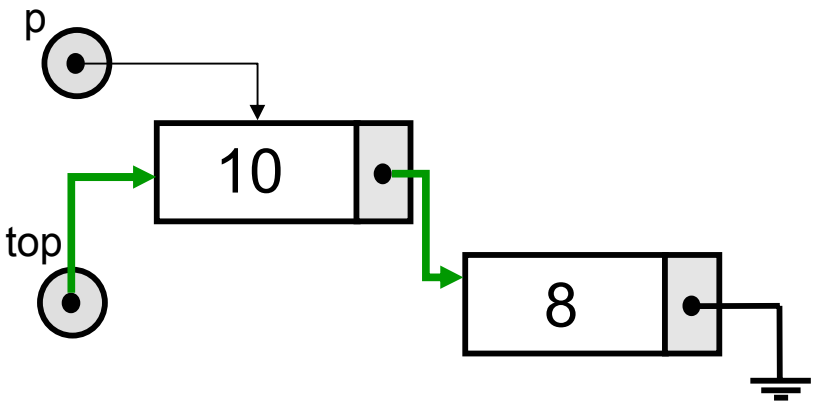



```
void Stack::Push(int x)
{  StackPointer p;

   p = new StackNode;
   if(p == NULL)
   { cout << “Memória insuficiente”;
     abort();
   }

   p->Entry = x;
   p->NextNode = top;
   ...
}
```

Operações Básicas: Push

void Stack::Push(int x)

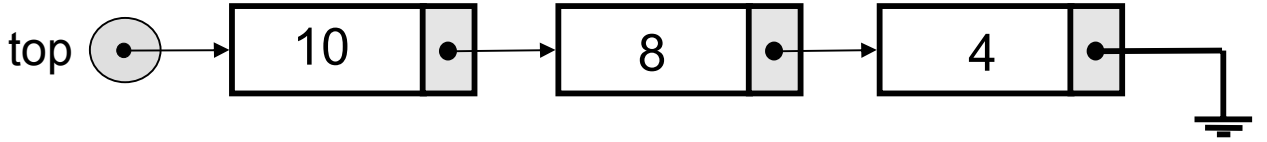
Caso contrário, colocamos o elemento **x** no campo de dados de **p** e alteramos os ponteiros




```
void Stack::Push(int x)
{  StackPointer p;

   p = new StackNode;
   if(p == NULL)
   { cout << “Memoria insuficiente”;
     abort();
   }

   p->Entry = x;
   p->NextNode = top;
   top = p;
}
```

Operações Básicas: Pop

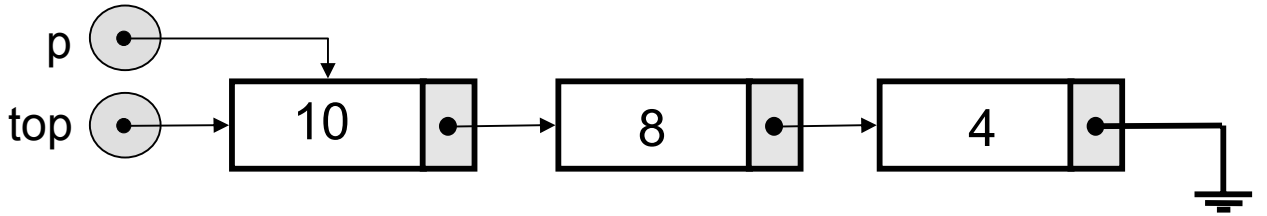
- Para retirar um elemento da pilha...



Operações Básicas: Pop

- Para retirar um elemento da pilha...

- ...usamos um ponteiro auxiliar **p**

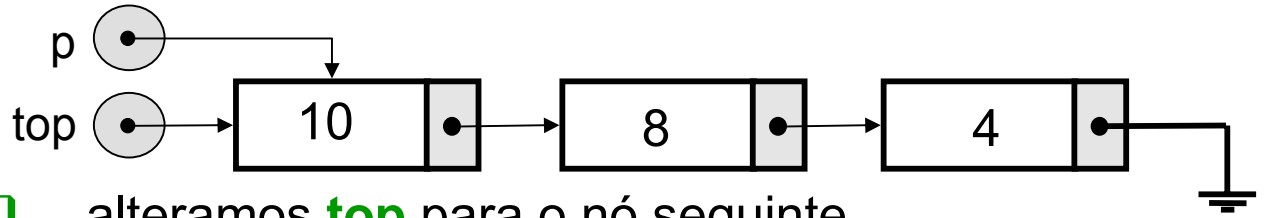



```
p = top;
```

Operações Básicas: Pop

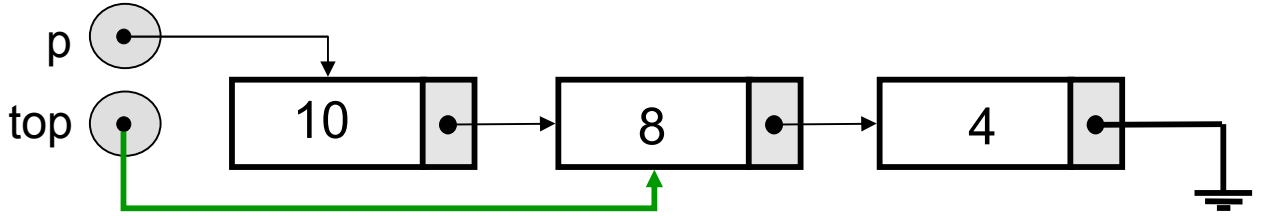
- Para retirar um elemento da pilha...

- ...usamos um ponteiro auxiliar **p**



p = top;

- ...alteramos **top** para o nó seguinte...

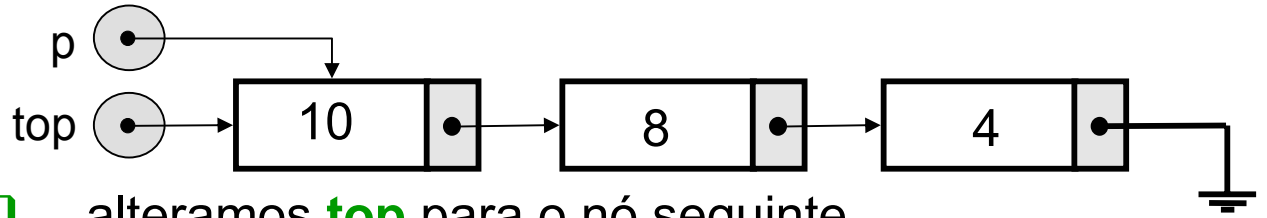


top = top->NextNode

Operações Básicas: Pop

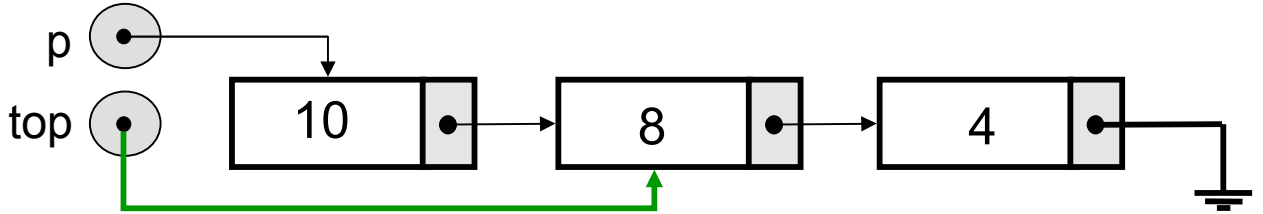
- Para retirar um elemento da pilha...

- ...usamos um ponteiro auxiliar **p**



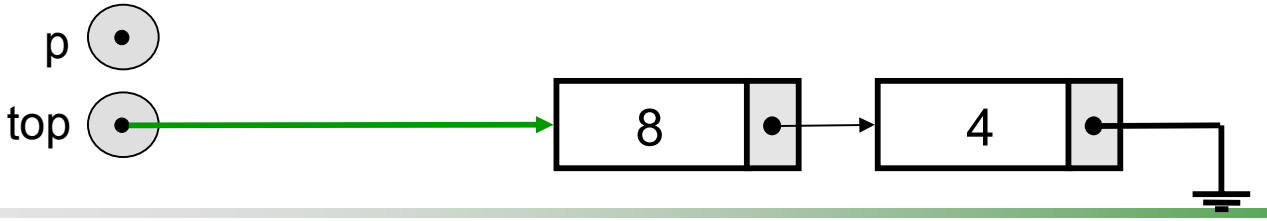
p = top;

- ...alteramos **top** para o nó seguinte...



top = top->NextNode

- ...e, finalmente, liberamos o espaço apontado por **p**

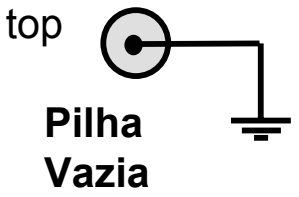


delete p;

Operações Básicas: Pop

void Stack::Pop(int &x)

Inicialmente, verificamos se a pilha está vazia. Em caso afirmativo, imprimimos uma mensagem de erro e terminamos

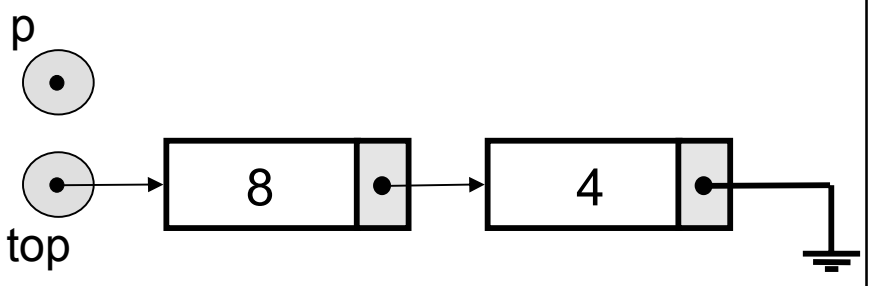


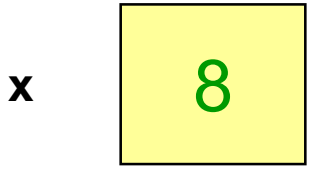

```
void Stack::Pop(int &x)
{ StackPointer p;

  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }
  ...



}
```

Operações Básicas: Pop

void Stack::Pop(int &x)

Caso contrário, armazenamos o valor do topo na variável **x**



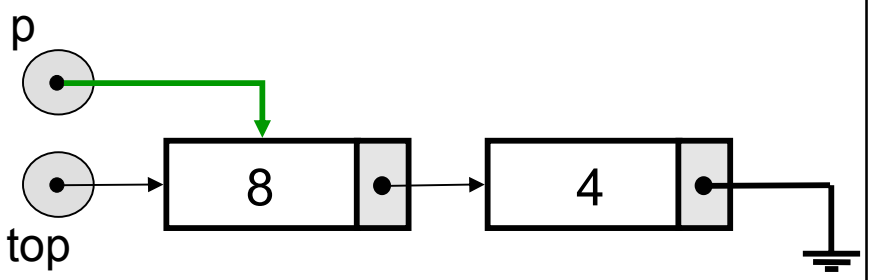


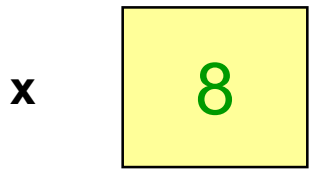

```
void Stack::Pop(int &x)
{ StackPointer p;

  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }

   x = top->Entry;
   ...


}
```

Operações Básicas: Pop

void Stack::Pop(int &x)

Apontamos o ponteiro auxiliar **p** para o topo da pilha...



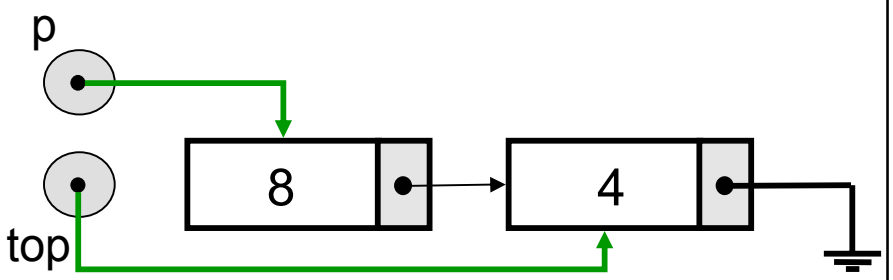


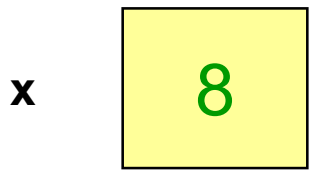

```
void Stack::Pop(int &x)
{ StackPointer p;

  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }

   x = top->Entry;
   p = top;
   ...

}
```

Operações Básicas: Pop

void Stack::Pop(int &x)

Apontamos o ponteiro auxiliar **p** para o topo da pilha; alteramos o topo para o próximo nó...



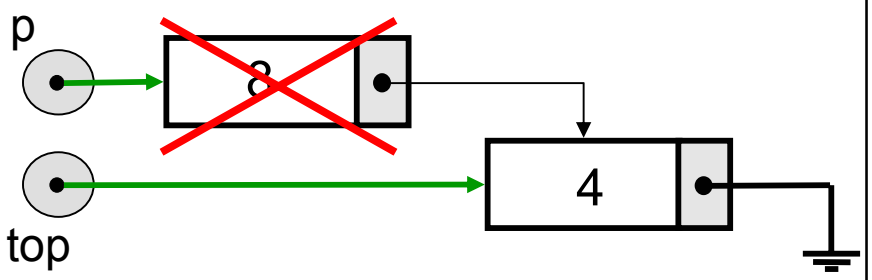



```
void Stack::Pop(int &x)
{ StackPointer p;

  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }

   x = top->Entry;
   p = top;
   top = top->NextNode;
   ...
}
```

Operações Básicas: Pop

void Stack::Pop(int &x)

Apontamos o ponteiro auxiliar **p** para o topo da pilha; alteramos o topo para o próximo nó e, finalmente, liberamos o espaço apontado por **p** (antigo topo)



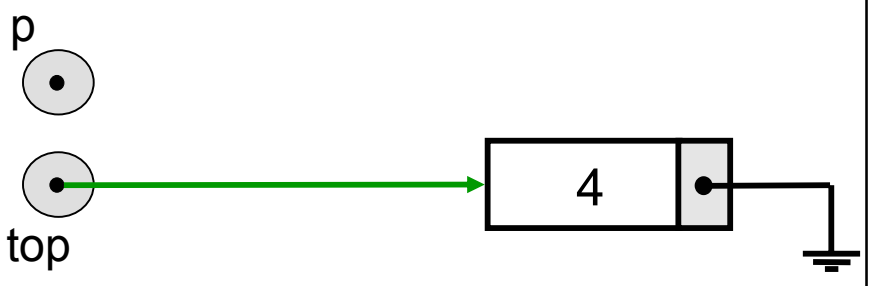
x 8

```
void Stack::Pop(int &x)
{ StackPointer p;

  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }

   x = top->Entry;
   p = top;
   top = top->NextNode;
   delete p;
}
```

Operações Básicas: Pop

void Stack::Pop(int &x)

Apontamos o ponteiro auxiliar **p** para o topo da pilha; alteramos o topo para o próximo nó e, finalmente, liberamos o espaço apontado por **p** (antigo topo)



x 8

```
void Stack::Pop(int &x)
{ StackPointer p;

  if (Empty())
   {  cout << “Pilha Vazia”;
      abort();
   }

   x = top->Entry;
   p = top;
   top = top->NextNode;
   delete p;
}
```

Outras Operações: Exercícios

- Defina a operação Clear utilizando apenas as operações Empty e Pop
- Defina a operação Clear utilizando diretamente ponteiros
- Defina a operação Top utilizando apenas Push e Pop e Empty
- Defina a operação Top utilizando diretamente ponteiros
- Defina a operação Size
  - Há alguma ineficiência em sua implementação? Em caso afirmativo, estenda o objeto de forma a torná-la mais eficiente
- Quais as vantagens e desvantagens de cada uma destas construções?

Solução Clear

- Usando apenas Empty e Pop

```
void Stack::Clear()
{ int x;

  while(! Empty())
    Pop(x);
}
```

- Utilizando campos do objeto

```
void Stack::Clear()
{ StackPointer p;

  while(top != NULL)
  { p = top;
    top = top->NextNode;
    delete p;
  }
}
```

Solução Top

- Usando apenas Push e Pop

```
void Stack::Top(int &x)
{  if(Empty())
   { cout << “Pilha vazia”;
     abort();
   }
   Pop(x);
   Push(x);
}
```

- Utilizando campos do objeto

```
void Stack::Top(int &x)
{  if(top == NULL)
   { cout << “Pilha vazia”;
     abort();
   }
   x = top->Entry;
}
```

Solução Size

```
int Stack::Size()
{ StackPointer p;
  int s=0;

  p = top;
  while(p != NULL)
  { s++;
    p = p->NextNode;
  }
  return s;
}
```

Pontos Importantes

- Nesta última parte, vimos uma forma de implementação de pilhas usando alocação dinâmica
- A vantagem de usar alocação dinâmica (quando comparada com a implementação utilizando vetores) é que não é necessário definir o tamanho da pilha, evitando desperdício de espaço
- Uma desvantagem deste tipo de implementação é que, a cada chamada de Push e Pop, memória deve ser alocada/liberada, o que pode tornar essa implementação, em geral, um pouco mais lenta do que a pilha contígua