



Tipos de Dados Abstractos

Algoritmos e Estruturas de Dados



Tipos Abstractos

- Um **tipo abstracto** é:
 - um tipo genérico de dados, dos quais não se conhece os valores
 - uma *interface* que define os acessos e as operações sobre os dados
 - O conjunto das *propriedades* algébricas da interface, que delimitam as possíveis implementações.
- Um programa que usa um tipo abstracto é um *cliente*. O cliente não tem acesso à implementação.
- Um programa que especifique o tipo de dados é uma **implementação**.



Pilha - *Stack*

- Operações abstractas:
 - inicialização;
 - colocação de um elemento na pilha;
 - remoção de um elemento (o último colocado); **LIFO**
 - indicar se pilha está vazia.
- interface para uma pilha que contenha inteiros:

```
public interface intStack{  
    public int empty();  
    public void push(int i);  
    public int pop();  
}
```

Usando a mesma interface,
podem ser definidas várias
implementações!



Pilha - *Stack*

- Operações abstractas:
 - inicialização;
 - colocação de um elemento na pilha;
 - remoção de um elemento (o último colocado); **LIFO**
 - indicar se pilha está vazia.

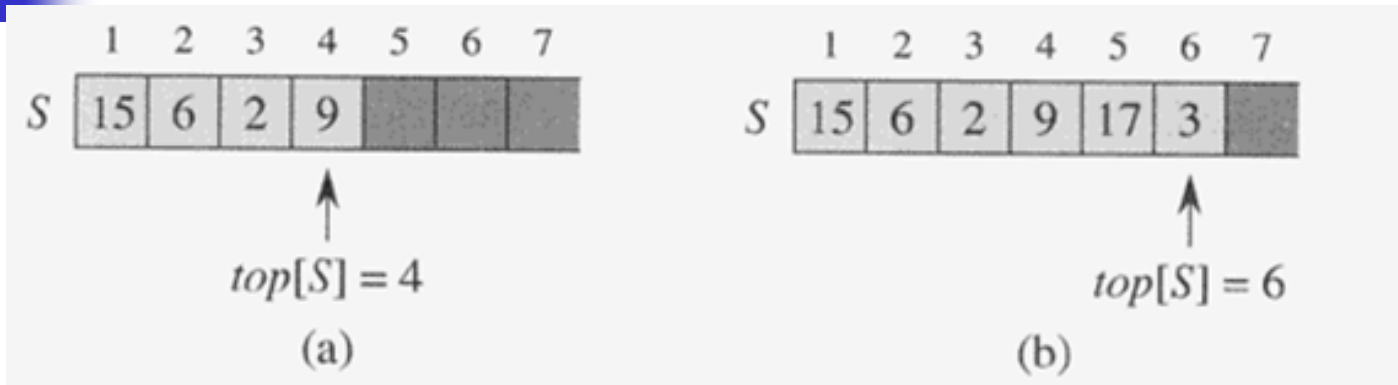
- interface para uma pilha genérica:

```
public interface Stack< E > {  
    public boolean isEmpty();  
    public E push(E i);  
    /*assume-se que antes de invocar os  
    métodos é testado se a pilha está vazia  
    */  
    public E pop();  
    public E peek();  
}
```

Usando a mesma interface,
podem ser definidas várias
implementações!

Cátia Vaz

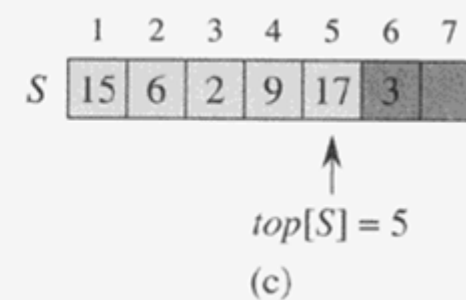
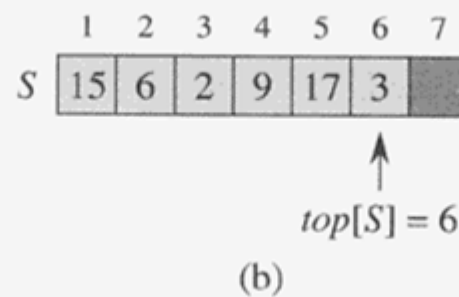
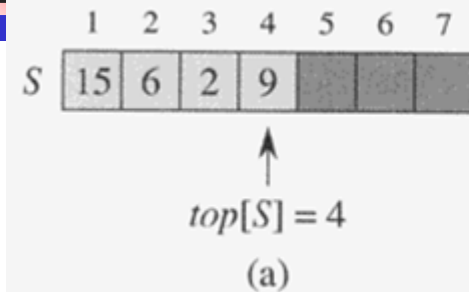
Pilha (*Stack*) - inserção



PUSH(S, x)

- 1 $top[S] \leftarrow top[S] + 1$
- 2 $S[top[S]] \leftarrow x$

Pilha (*Stack*) - remoção



```
POP( $S$ )  
1  if STACK-EMPTY( $S$ )  
2    then error "underflow"  
3  else  $top[S] \leftarrow top[S] - 1$   
4    return  $S[top[S] + 1]$ 
```

```
STACK-EMPTY( $S$ )  
1  if  $top[S] = 0$   
2    then return TRUE  
3  else return FALSE
```


Implementação do tipo pilha usando um *array*

```
//Não compila!!
public class StackArray<E> implements Stack<E>{
    private E[] s;
    private int top;
    public StackArray(int capacity) {
        s=new E[capacity];
    }
    //! a criação de um array genérico não é permitida em Java

    public boolean isEmpty(){ return top == 0; }
    //assume-se que antes de invocar o método pop e peek é testado se a pilha está vazia

    public E peek() { return s[top-1]; }
    public E push(E item) {
        if( top == s.length ) return null;
        s[top++] = item; return item;
    }
    public E pop() {
        E e = s[--top]; s[top] = null;
        return e;
    }
}
```

Vantagens: simplicidade.
Desvantagens: tamanho máximo limitado à partida.



Implementação do tipo pilha usando um *array*

```
public class StackArray<E> implements Stack<E>{
    private E[] s;
    private int top;
    public StackArray(int capacity) {
        s = ( E[] ) new Object[capacity]; //solução!
    }
    public boolean isEmpty(){ return top == 0; }
    //assume-se que antes de invocar o método pop e peek é testado se a pilha está vazia

    public E peek() { return s[top-1]; }
    public E push(E item) {
        if( top == s.length ) return null;
        s[top++] = item; return item;
    }
    public E pop() {
        E e = s[--top]; s[top] = null;
        return e;
    }
}
```

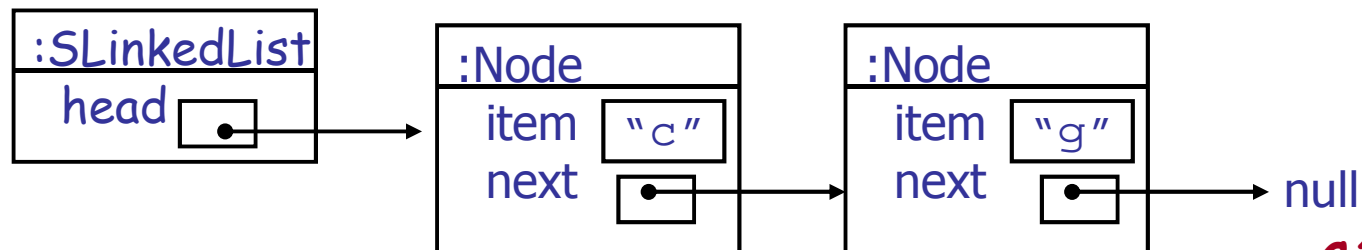
Vantagens: simplicidade.
Desvantagens: tamanho máximo limitado à partida.

Lista Simplesmente Ligada

- Uma **lista ligada** é uma colecção de objectos, designados por nós, em que cada um contém dados e uma referência para outro nó, de modo a formarem uma lista. O primeiro nó é designado por **head node**.
- Uma lista ligada diz-se **simplesmente ligada** quando cada nó contém uma referência para o próximo nó.

```
private class Node<E>{  
    private E data;  
    private Node next;  
    ...  
}
```

Exemplo:



Cátia Vaz

Implementação do tipo pilha usando um lista ligada

```
public class StackList<E> implements Stack<E>{
    private static class Node<E> {
        E item; Node<E> next;
        Node(E i, Node<E> n){ item=i; next = n}
    }
    private Node<E> head;
    public boolean isEmpty() {return head == null; }
    public E peek()          { return head.item; }
    public E push(E item) {
        head = new Node<E>(item, head);
        return item;
    }
    public E pop() {
        E item = head.item;
        head = head.next;
        return item;
    }
}
```

Vantagens:

- aumenta e diminui consoante se inserem ou removem novos elementos na pilha.

Desvantagens:

- ocupa mais memória.
- acesso mais lento.



Que estrutura de dados escolher?

- Depende :
 - das operações que sejam mais frequentes no algoritmo a implementar:
 - *Se a escolha for entre arrays e listas, dever-se-á escolher:*
 - *arrays* se forem predominantes as operações de **acesso e modificação**;
 - listas se forem predominantes as operações de **inserção e remoção**.
 - Usar **listas simples**, se estas operações ocorrem sempre no início.
 - Usar **listas simples com ponteiro adicional para o fim da lista**, se a inserção se fizer no fim e a remoção no início.
 - da **previsibilidade sobre a memória necessária**:
 - o uso de *arrays* pressupõe que se saiba que memória é suficiente;
 - quando não é previsível é necessário gerir a memória dinamicamente.



Iteradores

```
public interface Iterable<E>{  
    public Iterator<E> iterator();
```

```
public interface Iterator<E>{  
    public boolean hasNext();  
    public E next();  
    public void remove();
```



Implementação do iterador em StackList<E>

```
public class StackList<E> implements Stack<E>{
    //...
    public class StackIterator<E> implements Iterator<E>{

        //Nó dos dados retornados pelo next seguinte
        protected Node<E> node=head;

        public boolean hasNext(){ return node!=null;}

        //Retorna o elemento da posição actual e avança
        public E next(){
            if(node==null) throw new NoSuchElementException();
            E e=node.item;
            node=node.next; //Avança
            return e
        }

        //Neste exemplo de iterador não se permite a remoção através do iterador
        public void remove() { throw new UnsupportedOperationException();}
    }
}
```



Autoboxing

- Implementação da pilha genérica:
 - Permite objectos, não tipos primitivos.
- Tipo *Wrapper*:
 - Cada tipo primitivo tem um tipo de objecto *wrapper*!.
 - Ex: Integer é o tipo *wrapper* para int.
- Autoboxing: conversão automática entre um tipo primitivo e o seu *wrapper*.



Exercício

- Implemente uma aplicação que leia do standard input uma expressão aritmética em notação posfixa e calcule o valor da expressão. Considere que:
 - a expressão apenas envolve os operadores binários + e x;
 - os operadores e operandos são delimitados por um espaço.
 - Exemplo:
 - $5\ 9\ 8\ +\ 4\ 6\ *\ * 7\ +\ *$
é equivalente a:
 $(5 * ((9 + 8) * (4 * 6)) + 7)$
isto é, o seu valor é 2075



Fila-Queue

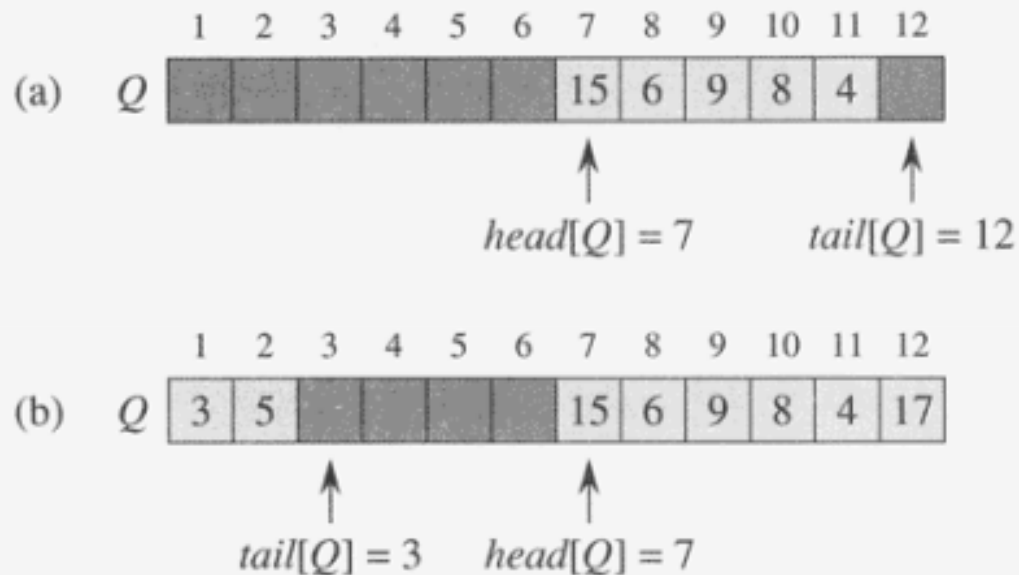
- Operações abstractas:
 - inicialização;
 - colocação de um elemento na fila;
 - remoção de um elemento (**FIFO**);
 - indicar se fila está vazia.

- interface para uma fila :

```
public interface Queue<E>{  
    public boolean isEmpty();  
    public boolean offer(E i);  
    public E poll();  
    public E peek();  
}
```

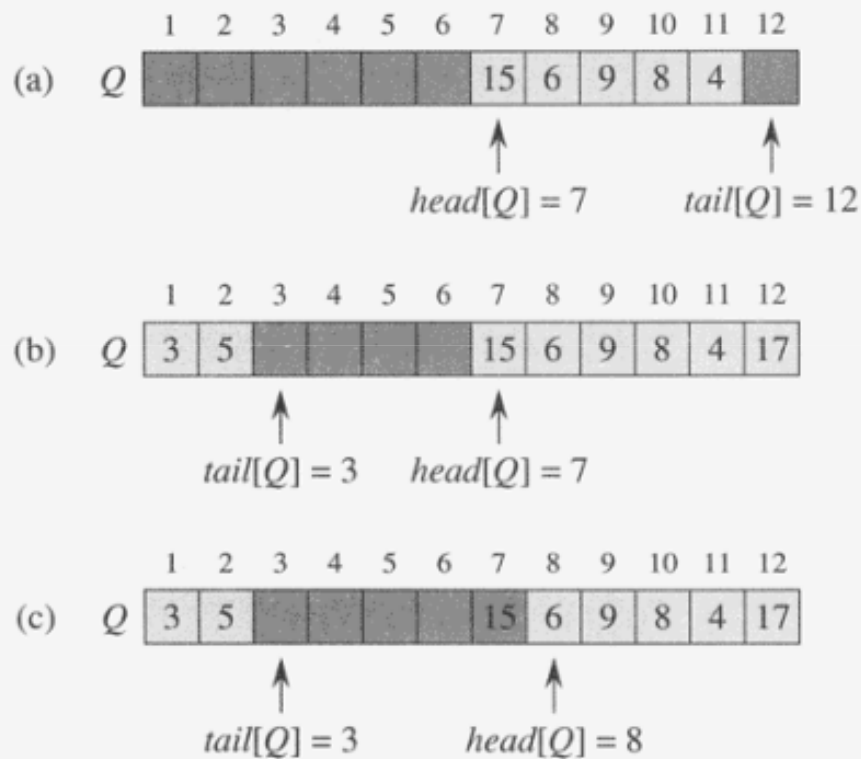
Usando a mesma interface,
podem ser definidas várias
implementações!

Fila (*Queue*) - inserção



```
ENQUEUE( $Q, x$ )  
1   $Q[tail[Q]] \leftarrow x$   
2  if  $tail[Q] = length[Q]$   
3      then  $tail[Q] \leftarrow 1$   
4      else  $tail[Q] \leftarrow tail[Q] + 1$ 
```

Fila (*Queue*) - remoção



DEQUEUE(Q)

```
1   $x \leftarrow Q[head[Q]]$ 
2  if  $head[Q] = length[Q]$ 
3      then  $head[Q] \leftarrow 1$ 
4      else  $head[Q] \leftarrow head[Q] + 1$ 
5  return  $x$ 
```

Implementação do tipo fila usando um *array*

```
public class QueueArray<E> implements Queue<E> {
    private E[] q;
    private int size, head, tail;
    public QueueArray(int maxN) { q=(E[])new Object[maxN]; }
    public boolean isEmpty() { return size == 0 ;}
    public boolean offer(E e){
        if( size == q.length) return false;
        q[tail] = e;
        tail = (tail+1) % q.length; ++size;
        return true;
    }
    private E remove() {
        E e= q[head]; q[head]= null;
        head= (head+1)%q.length; --size;
        return e;
    }
    public E poll(){ return ( size == 0 ) ? null : remove();
    //assume-se que antes de invocar o método get é testado se a fila está vazia
    public E peek() { return q[head]; }
}
```

Implementação do tipo fila usando uma lista ligada

```
public class QueueList<E> implements Queue<E> {  
    private static class Node<E> {  
        E item; Node<E> next;  
        Node() { next=this; }  
        Node(E i, Node<E> n) { item=i; next = n; }  
    }  
    private Node<E> tail = new Node<E>();  
    public boolean isEmpty() { return tail == tail.next; }  
    public boolean offer(E e){  
        tail.next = new Node<E>( e, tail.next ); tail = tail.next;  
        return true;  
    }  
    private E remove() {  
        Node<E> rem = tail.next.next; tail.next.next = rem.next;  
        if ( rem == tail ) tail = tail.next;  
        return rem.item;  
    }  
    public E poll() { return remove(); }  
    public E peek() { return tail.next.next.item; }  
}
```