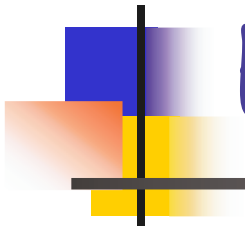
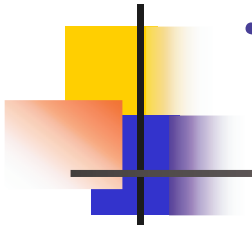


Estruturas de Dados Elementares

Algoritmos e Estuturas de Dados

Inverno 2006

Cátia Vaz

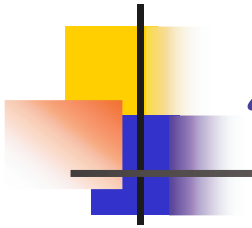


Tipologia de Dados

- Simples:
  - inteiros;
  - reais;
  - caracteres;
  - booleanos.
- Estruturados:
  - arrays;
  - listas;
  - árvores
  - listas de listas;
  - listas de arrays;
  - array de listas;
  - ...

Que estrutura de dados escolher?

- Depende :
  - **das operações que sejam mais frequentes no algoritmo a implementar:**
    - deverá escolher-se *arrays* se forem predominantes as operações de **acesso** e **modificação**;
    - deverá escolher-se 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.

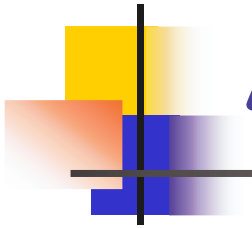


Arrays

- Sintaxe para criar um array

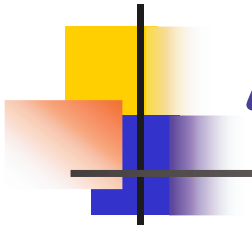
```
TipoBase[] nomeArray = new TipoBase[dimensão];
```

- O tipo dos elementos é denominado ***tipo base***.
- O tipo base do *array* pode ser de qualquer tipo.
- O número de elementos é a **dimensão** do *array*



Arrays

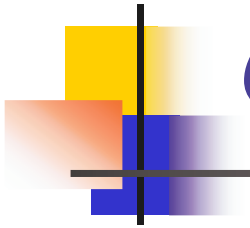
- O *array* tem uma única variável de instância pública, chamada **`length`**, que armazena o número de elementos que o *array* pode conter
- Os índices do *array* começam em 0 e terminam em **`nomeArray.length-1`**.
- O array pode ser iniciado quando é criado.
  - Exemplo: `double[] a = {3,3, 15.8, 9.7};`
- Os elementos de um ***array*** não iniciado são colocados com **o valor por omissão do tipo base**.



Arrays

- *Arrays* multi-dimensionais são implementados em Java usando *arrays* unidimensionais.
  - Exemplo: `int[][] nome_array = new int[nLinhas][nColunas];`
    - O seu tipo base é `int[]`.
    - Por isso, é um *array* de *arrays*.
- Linhas diferentes podem ter um número de colunas diferentes (***ragged arrays***).
  - Exemplo: `int[][] a=new int[3][];`

```
a[0]=new int[5];a[1]=new int[6]; a[2]=new int[7]
```



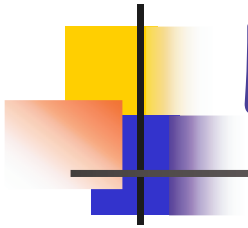
Crivo de Eratosthenes

- O objectivo é determinar todos os primos menores que um dado N.
  - array de booleanos **`primo[]`** de tamanho **`N`**;
  - Inicialização: **`primo[i]=true`** para **`2<=i<N`**.
  - Interpretação:
    - atribui o valor **`false`** aos elementos de **`primo`** que correspondem a índices que são conhecidos serem não primos (múltiplos de primos conhecidos).
    - Se **`primo[i]`** ainda tiver o valor **`true`** após ser atribuído **`false`** a todos os múltiplos de primos menores que **`i`**, então **`i`** é primo.

Crivo de Eratosthenes

```
import java.util.Scanner;
public class Crivo {
  public static void crivoEratosthenes(boolean primo[]){
    int n=primo.length;
    for (int i = 2; i < n; i++) primo[i] = true;
    for (int i = 2; i < n; i++)
      if (primo[i])
           for (int j = i; j*i < n; j++)
              primo[i*j] = false;
   }

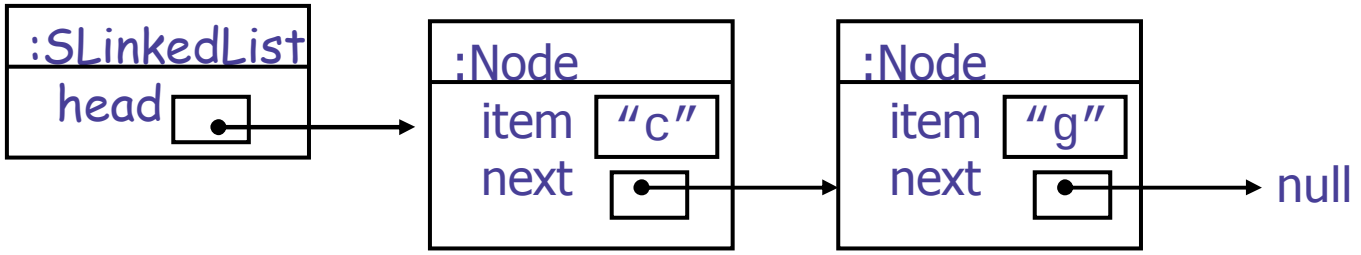
  public static void main(String[] args) {
    Scanner s=new Scanner(System.in);
    System.out.println("Introduza um numero inteiro positivo");
    int n=s.nextInt(); boolean primo[]=new boolean[n];
    crivoEratosthenes(primo);
    for(int i=0;i<n;i++)
        if(primo[i]) System.out.println(i +" e primo");
   }
}
```



Lista Simplesmente Ligada

- Uma lista ligada é uma colecção de objectos, designados por nós , em que cada um contêm 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{
        private E data;
        private Node next;
         ...
   }
  ```
  ```

Lista Simplesmente Ligada



- Exemplo de uma instância da classe **`LinkedList`**.
- Note-se que se está a adicionar objectos do tipo **`String`**.
- Exemplo:**`SLinkedList.java`**

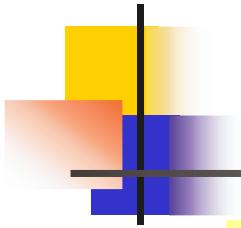
SLinkedList

```
public class SLinkedList<E> {
    private int size;
    private Node head;

    public SLinkedList(){size=0; head=null;}

    public int getSize(){. . .}
    public void addFirst(E e){. . .}
    public E removeFirst(){ . . .}
    public void addLast(E e){. . .}
    public E removeLast(){. . .}
    public String toString(){. . .}

/*Classe interna Node*/
     private class Node{...}
}
```



SLinkedList

```
private class Node{
   E item;
   Node next;

   Node(){
     item=null;
     next=null;}
   Node(E item){
     this.item=item;
     next=null;
   }
}
```

```
public int getSize(){
   return size;
}

public void addFirst(E e){
   Node aux=new Node(e);
   if(head==null){
        head=aux;
   }
   else{
        aux.next=head;
        head=aux;
   }
   size++;
}
```

SLinkedList

```
public E removeFirst(){
   if(head==null) return null;
   E e=head.item;
   head=head.next;
   size--;
   return e;
   }
public void addLast(E e){
   if(head==null){addFirst(e);}
   else{
         Node novo=new Node(e);
         Node aux=head;
         while(aux.next!=null){
                  aux=aux.next;
         }
         aux.next=novo;
         size++;
   }
}
```

```
public E removeLast(){
   if(head==null)return null;
   Node aux=head;
   Node prev=null;
   E e;
   while(aux.next!=null){
         prev=aux;
         aux=aux.next;
         }
   e=aux.item;
   if(prev==null){head=null;}
   else{prev.next=null;}
   size--;
   return e;
   }
```

Cátia Vaz

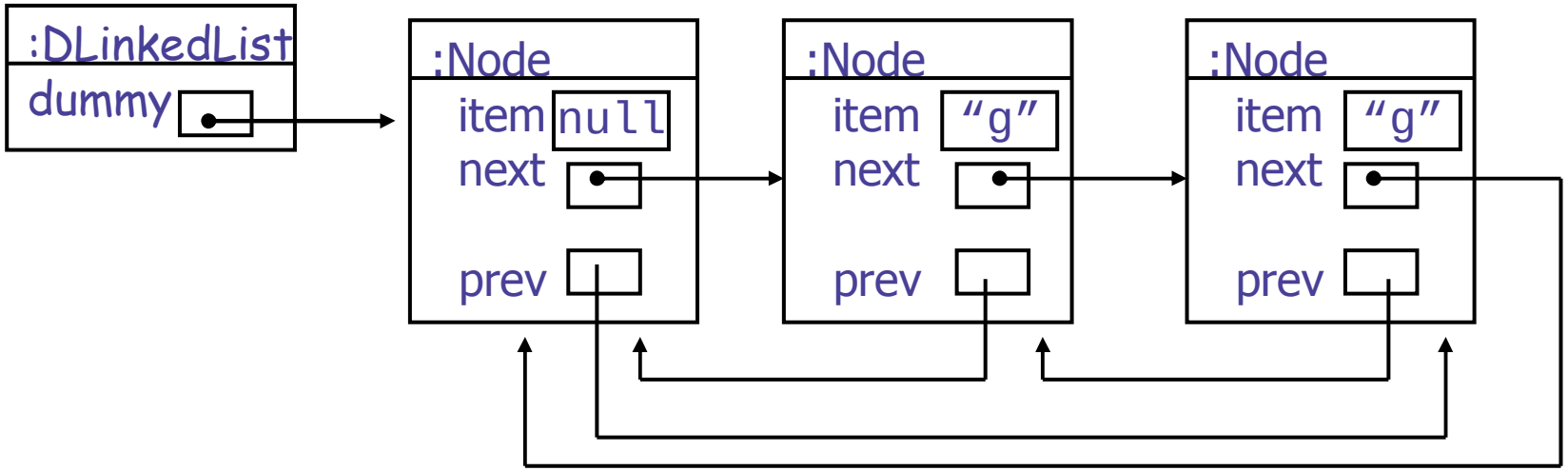
SLinkedList

```
public String toString(){
   StringBuffer sb = new
        StringBuffer("[");
   Node node=head;
   while(node!=null) {
      sb.append(node.item);
      node=node.next;
      if(node!=null){
         sb.append(", ");
        }
   }
   sb.append(']');
   return sb.toString();
}
```

```
private Node reverse(Node x){
   Node novoAnterior,actual=x;
   Node novoProximo=null;
   while(actual!=null){
      novoAnterior=actual.next;
      actual.next=novoProximo;
      novoProximo=actual;
      actual=novoAnterior;
   }
      return novoProximo;
   }

public void reverse(){
    this.head=reverse(this.head);
   }
```

Lista Duplamente Ligada Circular



- Note-se que se está a adicionar objectos do tipo **`String`**.
- Exemplo:**`DLinkedList.java`**

DLinkedList

```
public class DLinkedList<E> {
    private int size;
    private Node dummy;

    public DLinkedList(){size=0; dummy=new Node();}

    public int getSize(){. . .}
    public void add(E e){. . .}
    public boolean remove(E e){ . . .}
    public String toString(){. . .}

/*Classe interna Node*/
     private class Node{...}
}
```

DLinkedList

```
private class Node{
   private Node next, prev;
   private E item;
   public Node(){next=prev=this;}
   /*para node dummy; o atributo
    *obj fica a null*/

   public Node(E e, Node n){
      item=e;
      this.next=n;
      this.prev=n.prev;
      this.prev.next=this;
      n.prev=this;}
   /*n refere o node que vai ficar
    *a seguir a this*/

   public void remove() {
      this.next.prev=this.prev;
      this.prev.next=this.next; }
     }
```

```
public int getSize(){
   return size;
}

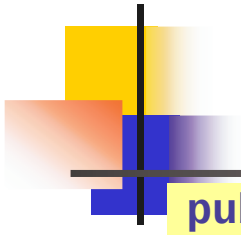
public boolean add(E e) {
   new Node(e, dummy );
   ++size;
   return true; }

public boolean remove (E e){
   Node aux=dummy.next;
   while(aux!=dummy){
      if(aux.item.equals(e)){
           aux.remove();
           return true;
        }
      aux=aux.next;
   }
   return false;
}
```

Cátia Vaz

DLinkedList

```
public String toString(){
    StringBuffer sb = new
         StringBuffer("[");
    Node node=dummy.next;
    while(node!=dummy) {
        sb.append(node.item);
        node=node.next;
        if(node!=dummy){
          sb.append(", "); }
    }
    sb.append(']');
    return sb.toString();
}
```



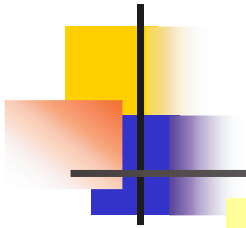
DCLinkedList

```
public class DCLinkedList<E extends Comparable<E>>{
    private int size;
    private Node dummy;

    public DCLinkedList(){size=0; dummy=new Node();}

    public int getSize(){. . .}
    public void insertionSort(E e){. . .}
    public boolean remove(E e){ . . .}
    public String toString(){. . .}

/*Classe interna Node*/
     private class Node{...}
}
```



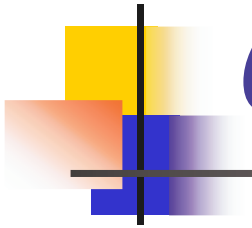
DCLinkedList

```
private class Node{
   private Node next, prev;
   private E item;
   public Node(){next=prev=this;}
   /*para node dummy; o atributo
    *obj fica a null*/

   public Node(E e, Node n){
      item=e;
      this.next=n;
      this.prev=n.prev;
      this.prev.next=this;
      n.prev=this;}
   /*n refere o node que vai ficar
    *a seguir a this*/

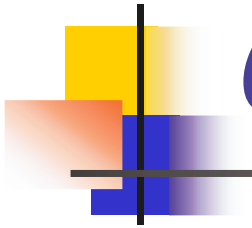
   public void remove() {
      this.next.prev=this.prev;
      this.prev.next=this.next; }
     }
```

```
public void insertionSort(E e){
   Node aux=dummy.next;
   while(aux!=dummy){
     if(aux.item.compareTo(e)>0){
          new Node(e,aux);
          size++;
          return;
        }
     aux=aux.next;
   }
   new Node(e,dummy);
   size++;
}
```



Grafos

- Os **Grafos** são estruturas de dados utilizados na definição de vários problemas computacionais.
- Existem dois tipos de grafos, **dirigidos** ou **não dirigidos**.
- Um **grafo dirigido** ***G*** é um par **(V,E)**, onde **V** é um conjunto finito e **E** é uma relação binária em **V**, tal que $E \subseteq V \times V$.
  - O conjunto **V** representa o conjunto de vértices do grafo;
  - *E* representa o conjunto de **arcos do grafo** ***G***, em que cada arco liga dois vértices do grafo.

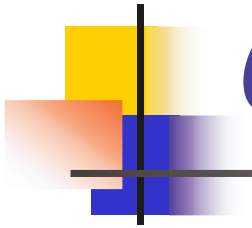


Grafos

- Uma forma de representar um grafo é utilizar um **array bidimensional**, que se designa por **matriz de adjacências**.
- Numa matriz de adjacências `m`, existe um arco do vértice i para o vértice j se `m[i][j]=true`.
- No caso de ser um **grafo não dirigido**, a matriz de adjacências é **simétrica**.

Grafos

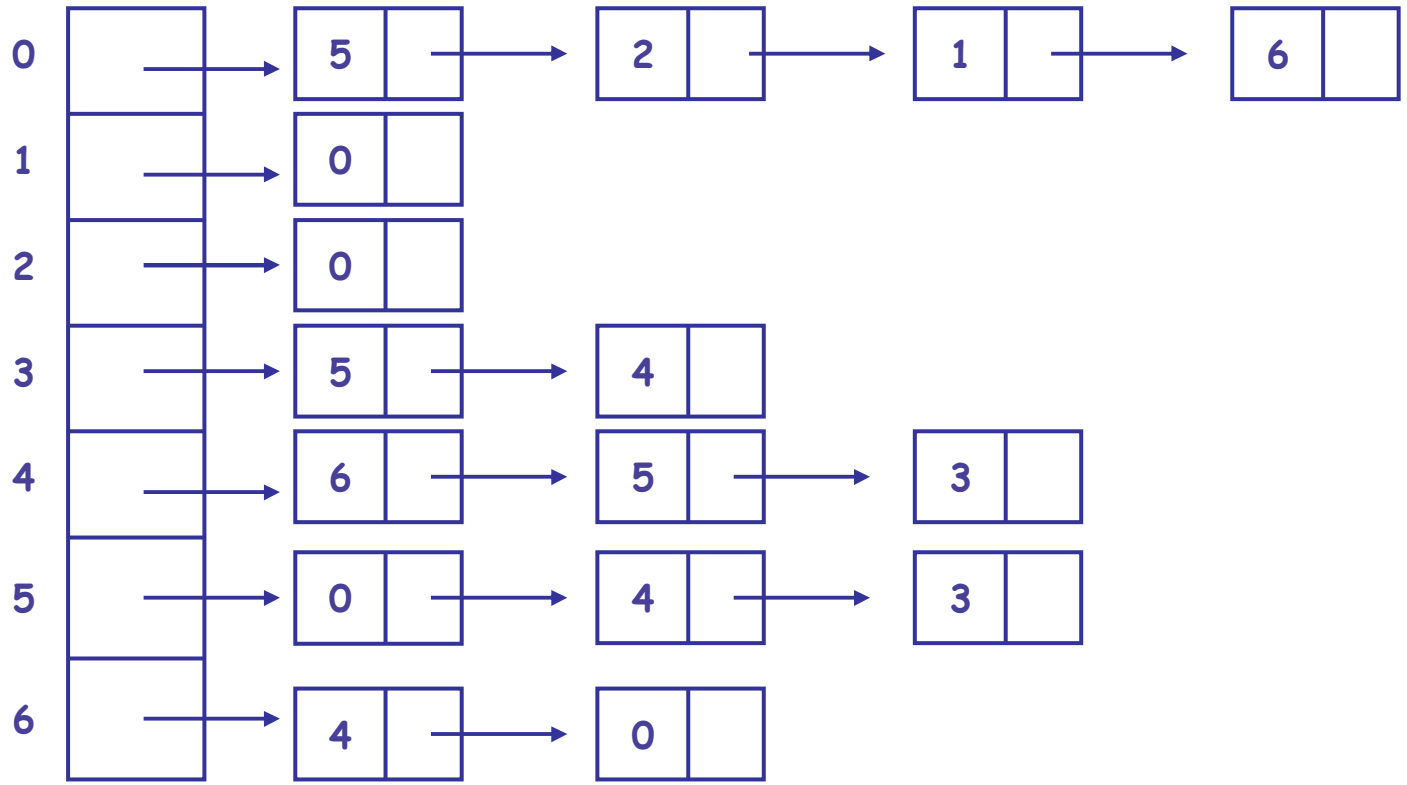
```
import java.util.Scanner;
public class MatrizAdjacencias {/*de um grafo não dirigido*/
   public static void main(String[] args){
        Scanner s=new Scanner(System.in);
        System.out.println("Introduza o numero de vertices");
        int v = s.nextInt();
        System.out.println("Introduza o numero de arcos");
        int a = s.nextInt();
        boolean adj[][] = new boolean[v][v];
        for (int i = 0; i < v; i++) adj[i][i] = true;
        int i,j;
        for(int k=0;k<a;i++){
           System.out.println("Introduza um novo arco: ");
           i=s.nextInt(); j=s.nextInt();
           adj[i][j] = true; adj[j][i] = true;
        }
  } }
```



Grafos

- Outra forma de representar um grafo é utilizar um **array de listas ligadas**, que se designa por **listas de adjacências**.
- Cada vértice tem associado a si uma lista ligada, na qual existe um nó para cada vértice que esteja conectado a esse vértice.
- Para grafos não dirigidos, se existir um nó para o vértice `j` na lista associada ao vértice `i`, então tem de existir um nó para o vértice `i` na lista associada ao vértice `j`.

Grafos



Cátia Vaz

Grafos

```
import java.util.Scanner;
public class ListaAdjacencias {
    private static class Node{
          int v;
          Node next;
          Node (int v, Node t){ this.v = v; next = t; }
     }
    public static void main(String[] args){
       Scanner s=new Scanner(System.in);
       System.out.println("Introduza o numero de vertices");
       int v = s.nextInt();
       System.out.println("Introduza o numero de arcos");
       int a = s.nextInt();
          Node adj[]=new Node[v]; int i,j;
          for(int k=0;k<a;k++){
             System.out.println("Introduza um novo arco: ");
             i=s.nextInt(); j=s.nextInt();
             adj[j] = new Node(i, adj[j]); adj[i] = new Node(j, adj[i]);
          }
  } }
```