Algorithms and Data Structures

Alfred Strohmeier

alfred.strohmeier@epfl.ch
http://lglwww.epfl.ch

March 2000

Swiss Federal Institute of Technology in Lausanne
Department of Computer Science
Software Engineering Laboratory

Table of Contents

I. Resources

Presentation Sequence

Bibliography

Bibliography

Aho A.V.; Hopcroft J.E.; Ullman J.D.; The Design and Analysis of Computer Algorithms; Addison-Wesley, 1974.

Aho A.V.; Hopcroft J.E.; Ullman J.D.; Data Structures and Algorithms; Addison-Wesley, 1983.

Booch G.; Software Components with Ada; Benjamin/Cummings, 1987.

Cormen T.H.; Leiserson C.E.; Riverst R.L.; Introduction to Algorithms; The MIT Press, 1991.

Feldman M.B.; Data Structures with Ada; Prentice-Hall, 1985.

Garey M.R.; Johnson D.S.; Computers and Intractability: A guide to NP-Completeness; W.H.Freeman, 1979.

Bibliography

Gonnet G.H.; Handbook of Algorithms and Data Structures; Addison-Wesley, 1984.

Gonnet G.H.; Baeza-Yates R.; Handbook of algorithms and data structures: in Pascal and C (2nd ed.); Addison-Wesley, 1991.

Horowitz E.; Sahni S.; Fundamentals of Data Structures in Pascal; Computer Science Press, 3rd ed., 1990.

Kernighan B.W.; Plauger P.J.; Software Tools in Pascal; Addison-Wesley, 1981.

Kingston J.H.; Algorithms and Data Structures; Addison-Wesley, 1990.

Bibliography

Knuth D.E; The Art of Computer Programming;

Vol.1: Fundamental Algorithms; 1973 (2nd edition);

Vol.2: Seminumerical Algorithms; 1981 (2nd edition);

Vol.3: Sorting and Searching; 1975 (2nd printing);

Addison-Wesley.

Kruse R.L.; Data Structures and Program Design; Prentice Hall, 1987 (2nd edition).

Mehlhorn K.; Data Structures and Algorithms;

Vol.1: Sorting and Searching;

Vol.2: Graph algorithms and NP-completeness;

Vol.3: Multi -dimensional Searching and computational geometry;

Springer; 1984.

Bibliography

Moret B.M.E.; Shapiro H.D.; Algorithms from P to NP;

Vol.1: Design and Efficiency; 1991;

Vol.2: Heuristics and Complexity;

Benjamin/Cummings.

Reingold E.M.; Nievergelt J.; Deo N.; Combinatorial Algorithms: Theory and Practice; Prentice-Hall, 1977.

Riley J.H.; Advanced Programming and Data Structures Using Pascal; PWS-Kent Publishing Co., Boston, 1990.

Sedgewick R.; Algorithms; Addison-Wesley, 1988 (2nd edition).

Stubbs D.F.; Webre N.W.; Data Structures with Abstract Data Types and Pascal; Brooks/Cole, 1985.

Bibliography

Tarjan R.E.; Data Structures and Network Algorithms; SIAM, 1983.

Tenenbaum A.M.; Augenstein M.J.; Data Structures using Pascal; Prentice-Hall, 1981.

Uhl J.; Schmid H.A.; A Systematic Catalogue of Reusable Abstract Data Types; Springer, 1990.

Wilf H.S.; Algorithms and Complexity, Prentice-Hall, 1986.

Wirth N.; Algorithms and Data Structures; Prentice-Hall,1986.

Wirth N.; Algorithms + Data Structures = Programs; Prentice-Hall, 1976.

Main references used for the classes are in bold.

II. Sorting

List of main sorting techniques
Performance comparison
Specification of a generic sort procedure
Use of the generic sort procedure
Selection sort
Insertion sort
Bubble sort
Quick sort

Sorting Techniques

Selection Sort
- Straight Selection Sort
- Quadratic Selection Sort

Insertion Sort
- Straight (Linear) Insertion Sort
- Binary Insertion Sort
- Shell Sort

Exchange Sort
- Straight Exchange Sort (Bubble Sort)
- Shaker Sort
- Quick Sort
- Radix Sort

Tree Sort
- Binary Tree Sort
- Heap Sort

Merge Sort

External Sorting
- Sort-Merge
- Polyphase Merge

Table of Comparison of Performance of Sorting Techniques

(see additional file)

Specification of Generic Sort

```
generic

   type Element_Type is private;

   with function "<"
         (Left, Right: Element_Type)
         return Boolean;

   type Index_Type is (<>);

   type Table_Type is array
         (Index_Type range <>)
         of Element_Type;

procedure Sort_G
         (Table: in out Table_Type);
-- Sort in increasing order of "<".
```

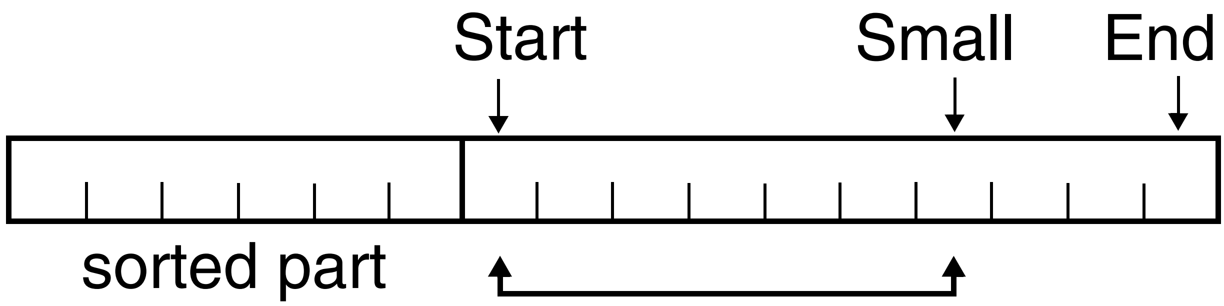
Use of Generic Sort

```
with Sort_G, Ada.Text_IO;
procedure Sort_Demo is

   procedure Sort_String is new Sort_G
      (Element_Type => Character,
      "<" => "<",
      Index_Type => Positive,
      Table_Type => String);

   My_String: String (1..6) := "BFCAED";

begin -- Sort_Demo
   Ada.Text_IO. Put_Line
         ("Before Sorting: " & My_String);
   Sort_String (My_String);
   Ada.Text_IO. Put_Line
         ("After Sorting: "& My_String);
end Sort_Demo;
```

Selection Sort

Principle

Basic operation: Find the smallest element in a sequence, and place this element at the start of the sequence.



Basic Idea:

- Find the index Small;
- Exchange the values located at Start and Small;
- Advance Start.

Sorting Table (Start .. End):

- Find Small in Start .. End;
- Exchange Table (Start) and Table (Small);
- Sort Table (Start + 1 .. End);

Selection Sort

Example

390	205	182	45	235
45	205	182	390	235
45	182	205	390	235
45	182	205	390	235
45	182	205	235	390

Table 1: Selection Sort

Selection Sort

```
procedure Sort_G (Table: in out Table_Type) is
    Small: Index_Type;
begin
    if Table'Length <= 1 then
        return;
    end if;
    for I in Table'First..Index_Type'Pred (Table'Last) loop
        Small := I;
        for J in Index_Type'Succ (I)..Table'Last loop
            if Table (J) < Table (Small) then
                Small := J;
            end if;
        end loop;
        Swap (Table (I), Table (Small));
    end loop;
end Sort_G;
```

Selection Sort

Complexity

We will neglect the index operations. We will therefore count only operations on the elements of the sequence.

n is the length of the sequence.

The number of executions of the interior loop is:

$$(n-1) + (n-2) + ... + 1 = (1/2)*n*(n-1)$$

The interior loop contains one comparison.

The exterior loop is executed n-1 times.

The exterior loop contains one exchange.

Number de comparisons: $(1/2)*n*(n-1)$

Number of exchanges: $n-1$

Selection Sort

Assessment

The effort is independent from the initial arrangement.

Negative: $O(n^2)$ comparisons are needed, independently of the initial order, even if the elements are already sorted.

Positive: Never more than $O(n)$ moves are needed.

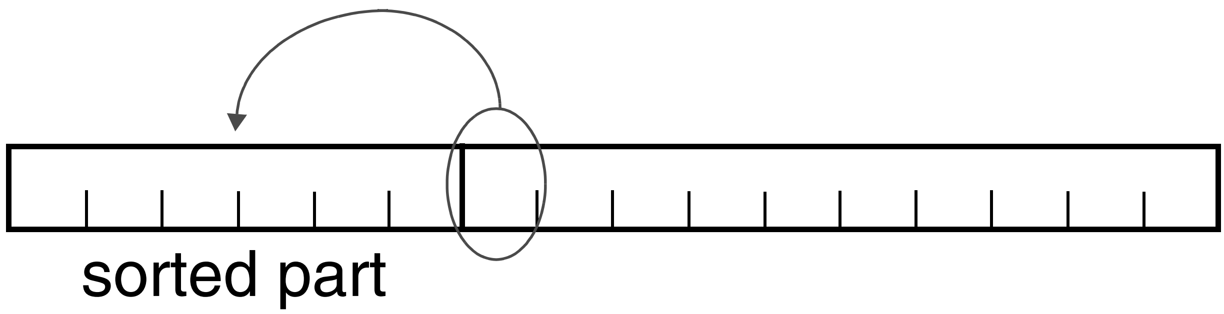
Conclusion: It’s a good technique for elements heavy to move, but easy to compare.

Insertion Sort

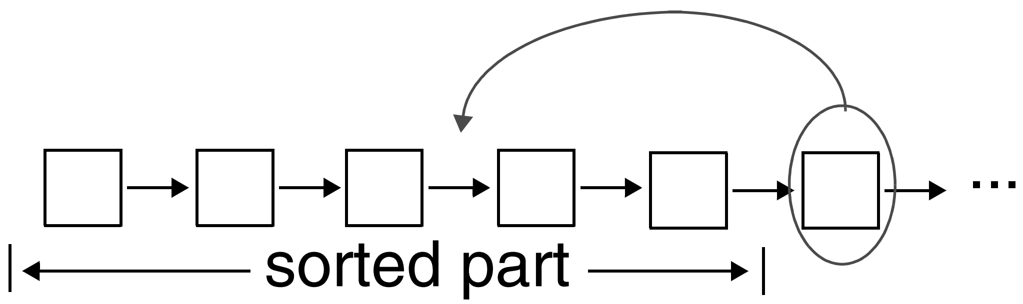
Principle

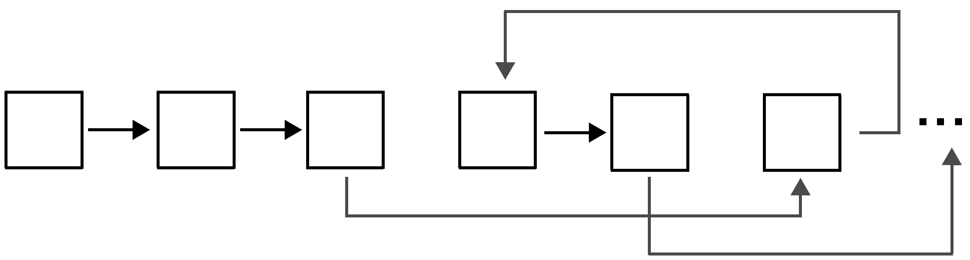
Basic operation: Insert an element in a sorted sequence keeping the sequence sorted.

Array



Linked List





Insertion Sort

Example: Exterior Loop

205	45	390	235	182
45	205	390	235	182
45	205	390	235	182
45	205	235	390	182
45	182	205	235	390

Table 2: Insertion Sort, Exterior Loop

Insertion Sort

Example: Interior Loop, moving the last element (I=5, Temp=182)

45	205	235	390	182
45	205	235	390	182
45	205	235	390	390
45	205	235	235	390
45	205	205	235	390
45	182	205	235	390

Table 3: Insertion Sort, Interior Loop

Insertion Sort

```
procedure Sort_G (Table : in out Table_Type) is
    Temp : Element_Type;
    J : Index_Type;
begin -- Sort_G
    if Table'Length <= 1 then
        return;
    end if;
    for I in Index_Type'Succ (Table'First) ..Table'Last loop
        Temp := Table (I);
        J := I;
        while Temp < Table (Index_Type'Pred (J)) loop
            Table (J) := Table (Index_Type'Pred (J));
            J := Index_Type'Pred (J);
            exit when J = Table'First;
        end loop;
        Table (J) := Temp;
    end loop;
end Sort_G;
```

Insertion Sort

Complexity

n is the length of the sequence

The exterior loop is executed n-1 times.

Interior loop:

Best case: 0

Worst case: 1+2+...+(n-1) = (1/2)*n*(n-1)

On average: One must walk through half of the list before finding the location where to insert the element: (1/4)*n*(n-1)

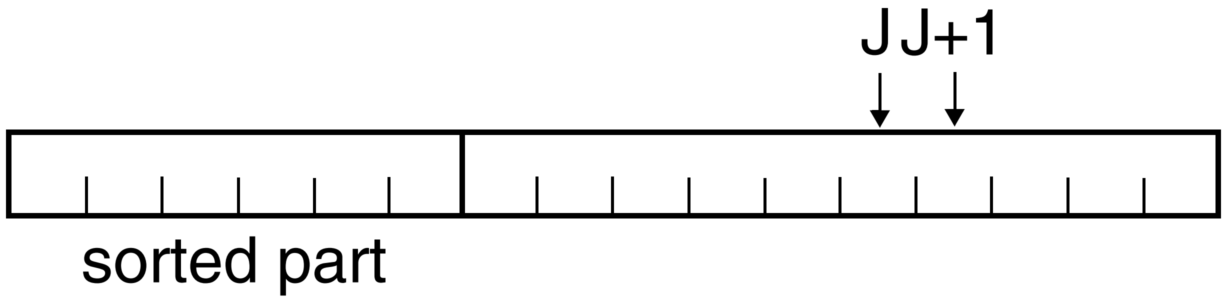
	Comparisons	Exchanges
Best Case	n-1	2*(n-1)
Average	(1/4)*n*(n-1)	(1/4)*n*(n-1) + 2*(n-1)
Worst Case	(1/2)*n*(n-1)	(1/2)*n*(n-1) + 2*(n-1)

Table 4: Performance of Insertion Sort

Bubble Sort, or Straight Exchange Sort

Principle

Basic Operation: Walk through the sequence and exchange adjacent elements if not in order.



Basic idea:

- walk through the unsorted part from the end;
- exchange adjacent elements if not in order;
- increase the sorted part, decrease the unsorted part by one element.

Bubble Sort

Example: First Pass

390	205	182	45	235
390	205	182	45	235
390	205	45	182	235
390	45	205	182	235
45	390	205	182	235

Table 5: Bubble Sort

Bubble Sort

Example: Second Pass

45	390	205	182	235
45	390	205	182	235
45	390	182	205	235
45	182	390	205	235

Table 6: Bubble Sort

Bubble Sort

Example: Third Pass

45	182	390	205	235
45	182	390	205	235
45	182	205	390	235

Table 7: Bubble Sort

Example: Fourth Pass

45	182	205	390	235
45	182	205	235	390

Table 8: Bubble Sort

Bubble Sort

No Sentinel

```
procedure Sort_G (Table: in out Table_Type) is
begin

    if Table'Length <= 1 then
        return;
    end if;

    for I in Table'First..Index_Type'Pred (Table'Last) loop

        for J in reverse Index_Type'Succ (I)..Table'Last loop
            if Table (J) < Table (Index_Type'Pred (J)) then
                Swap (Table (J), Table (Index_Type'Pred (J));
            end if;
        end loop;

    end loop;

end Sort_G;
```

Bubble Sort

With Sentinel

```
procedure Sort_G (Table: in out Table_Type) is
    Sorted: Boolean;
begin

    if Table'Length <= 1 then
        return;
    end if;

    for I in Table'First..Index_Type'Pred (Table'Last) loop
        Sorted := True;

        for J in reverse Index_Type'Succ (I)..Table'Last loop
            if Table (J) < Table (Index_Type'Pred (J)) then
                Sorted := False;
                Swap (Table (J), Table (Index_Type'Pred (J));
            end if;
        end loop;

        exit when Sorted;
    end loop;

end Sort_G;
```

Bubble Sort

Complexity

n is the length of the sequence.

k ($1 \leq k \leq n-1$)is the number of executions of the exterior loop (it is equal to the number of elements not in order plus one).

The number of executions of the body of the interior loop is:

- $(n-1) + (n-2) +... + (n-k) = (1/2)*(2n-k-1)*k$

The body of the interior loop contains:

- one comparison,
- sometimes an exchange.

Best case (ordered sequence):

- Number of comparisons: n-1
- Number of exchanges: 0

Worst case (inversely ordered sequence)

- Number of comparisons: (1/2)*n*(n-1)
- Number of exchanges: (1/2)*n*(n-1)

Average:

- Same magnitude as Worst Case.

Quick Sort

Principle

The Algorithm is recursive.

One step rearranges the sequence:

$a_1\ a_2 ....a_n$

in such a way that for some a_j, all elements with a smaller index than j are smaller than a_j, and all elements with a larger index are larger than a_j:

$a_1 \leq a_j \quad a_2 \leq a_j \quad ...\ a_{j-1} \leq a_j$

$a_j \leq a_{j+1} \quad a_j \leq a_{j+2} \quad ...\ a_j \leq a_n$

a_j is called the pivot.

Sorting Table (Start..End):

- Partition Table (Start..End), and call J the location of partitioning;
- Sort Table (Start..J-1);
- Sort Table (J+1..End).

Quick Sort

40	15	30	25	60	10	75	45	65	35	50	20	70
40	15	30	25	20	10	75	45	65	35	50	60	70
40	15	30	25	20	10	35	45	65	75	50	60	70
35	15	30	25	20	10	40	45	65	75	50	60	70

Table 9: Partitioning

Starting on each end of the table, we move two pointers towards the centre of the table. Whenever the element in the lower part is larger than the pivot and the element in the upper part is smaller, we exchange them. When the pointers cross, we move the pivot at that position.

Quick Sort: Sort_G (1)

```
procedure Sort_G (Table: in out Table_Type) is

    Pivot_Index: Index_Type;

    function "<=" (Left, Right: Element_Type)
        return Boolean is
    begin
        return not (Right < Left);
    end "<=";

    procedure Swap (X, Y: in out Element_Type) is
        T: constant Element_Type := X;
    begin
        X := Y; Y := T;
    end Swap;

    procedure Partition
        (Table: in out Table_Type;
        Pivot_Index: out Index_Type) is separate;
```

Quick Sort: Sort_G (2)

```
begin -- Sort_G
    if Table'First < Table'Last then

        -- Split the table separated by value at Pivot_Index
        Partition (Table, Pivot_Index);

        -- Sort left and right parts:
        Sort_G (Table
            (Table'First..Index_Type'Pred (Pivot_Index)));
        Sort_G (Table
            (Index_Type'Succ (Pivot_Index)..Table'Last));

    end if;
end Sort_G;
```

Quick Sort: Partition (1)

```
separate (Sort_G)
procedure Partition
        (Table: in out Table_Type;
        Pivot_Index: out Index_Type) is
    Up: Index_Type := Table'First;
    Down: Index_Type := Table'Last;
    Pivot: Table (Table'First);
begin
    loop

        -- Move Up to the next value larger than Pivot:
        while (Up < Table'Last)
            and then (Table (Up) <= Pivot) loop
                Up := Index_Type'Succ (Up);
        end loop;
        -- Assertion: (Up = Table'Last) or
                          (Pivot < Table (Up))

        -- Move Down to the next value less than or
                                                equal to Pivot:
        while Pivot < Table (Down) loop
            Down := Index_Type'Pred (Down);
        end loop;
        -- Assertion: Table (Down) <= Pivot.
```

Quick Sort: Partition (2)

```
        -- Exchange out of order values:
        if Up < Down then
            Swap (Table (Up), Table (Down));
        end if;
        exit when Up >= Down;
    end loop;
    -- Assertion: Table'First <= I <= Down =>
                                        Table (I) <= Pivot.
    -- Assertion: Down < I <= Down => Pivot < Table (I)

    -- Put pivot value where it has to be and
                                    define Pivot_Index:
    Swap (Table (Table'First), Table (Down));
    Pivot_Index  := Down;
end Partition;
```

Quick Sort

Complexity

Worst case: The sequence is already ordered.

Consequence: The partition is always degenerated.

Storage space:

The procedure calls itself n-1 times, and the requirement for storage is therefore proportional to n. This is unacceptable.

Solution: Choose for the pivot the median of the first, last and middle element in the table. Place the median value at the first position of the table and use the algorithm as shown (Median-of-Three Partitioning).

Execution time:

The execution of Partition for a sequence of length k needs k comparisons. Execution time is therefore proportional to n^2.

Quick Sort

Complexity

Best case: The sequence is always divided exactly at its mid-position.

Suppose $n = 2^m$

Quicksort for a sequence of size 2^m calls itself twice with a sequence of size 2^{m-1}.

Storage space:

$S_{2^m} = S_{2^{m-1}} + 1$

(the maximum for a recursive descent)

therefore:

$S_{2^m} \approx m$ and hence $S_n = O(\log n)$

Time behavior:

$C_{2^m} = 2C_{2^{m-1}} + 2^m$

(The 2^m elements must be compared with the pivot)

therefore:

$C_{2^m} \approx 2^m(m+1)$ and hence $C_n = O(n\log n)$

Quick Sort

Complexity

Average case: Same result as for the best case.

Idea about how to proceed for estimating the number of comparisons:

We consider a randomly selected permutation of n elements. The element at position k has a probability of 1/n to be the pivot. n-1 comparisons are needed for comparing the pivot with all the other elements. The recurrent relation is therefore:

$$c_0 = 1$$

$$c_n = n - 1 + \frac{1}{n} \cdot \sum_{k=1}^{n} (c_{k-1} + c_{n-k})$$

Quick Sort

Remarks

Parameter passing:

Beware of passing the Table parameter of Sort_G by copy!

Solution in Ada:

Write local procedures which use the index bounds of the table as parameters, and therefore work on the global variable Table.

Problem with recursion:

For "small tables" (between 5 and 25 elements), use an insertion sort.

Quick Sort is not stable!

III. Data Structures

List of the main data structures
Logical structure versus representation
Example: Subset
Various kinds of lists
Representations by lists
Abstract Data Type

Data Structures

Stack (Pile)

Queue (Queue, File d'attente)

- Deque (Double-Entry Queue,
  Queue à double entrée)
- Priority Queue (Queue de priorité)

Set (Ensemble)

Bag (Multiset, Multi-ensemble)

Vector (Vecteur)

Matrix (Matrice)

String (Chaîne)

(Linked) List (Liste chaînée)

- Linear List (Liste linéaire)
- Circular List (Liste circulaire)
- Doubly-linked List (Liste doublement chaînée)

Ring (Anneau)

Data Structures

Tree (Arbre)
- Ordered Tree (Arbre ordonné)
  - (children are ordered)
- 2-Tree (Arbre d'ordre 2)
  - (every node has 0 or 2 children)
- Trie (from re<u>trie</u>val)
  - (also called "Lexicographic Search Tree")
  - (a trie of order m is empty or is a sequence of m tries)

Binary Tree (Arbre binaire)
- Binary Search Tree (Arbre de recherche)
- AVL-Tree (Arbre équilibré)
- Heap (Tas)

Multiway Search Tree
- B-Tree

Data Structures

Graph (Graphe)
- Directed Graph (Graphe orienté)
- Undirected Graph (Graphe non orienté)
- Weighted Graph (Graphe valué)
- DAG (Directed Acyclic Graph, Graphe orienté acyclique)

Map (Mappe, Table associative)
- Hash Table (Table de hachage)

File (Fichier)
- Sequential File (Fichier sequentiel)
- Direct Access File (Fichier à accès direct)
- Indexed File (Fichier indexé, fichier en accès par clé)
- Indexed-Sequential File (ISAM) (Fichier indexé trié)

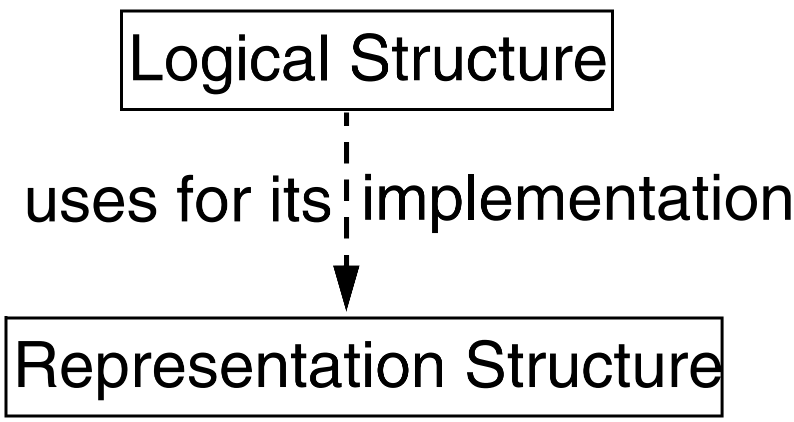
Representation of a Data Structure

It is important to distinguish between:

The data structure with its logical properties (ADT, abstract data type, type de données abstrait);

The representation of this data structure, or its implementation.

The representation of a data structure is usually also a data structure, but at a lower level of abstraction.



Subset

Representation

A subset E of a finite discrete set A can be represented by:

a) A characteristic function or a vector of booleans:

Membership: A $\longrightarrow$ {True, False}

e $\in$ E iff Membership(e)

b) A contiguous sequence that enumerates the elements belonging to the subset:

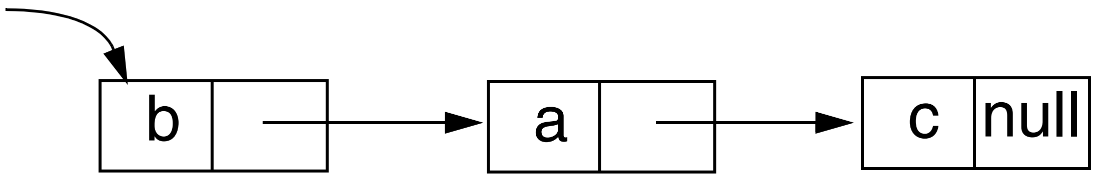
(V(i), i $\in$ [1, size(E)], V(i) $\in$ A)

e $\in$ E iff $\exists$ i $\in$ [1, size(E)] such that e = V(i)

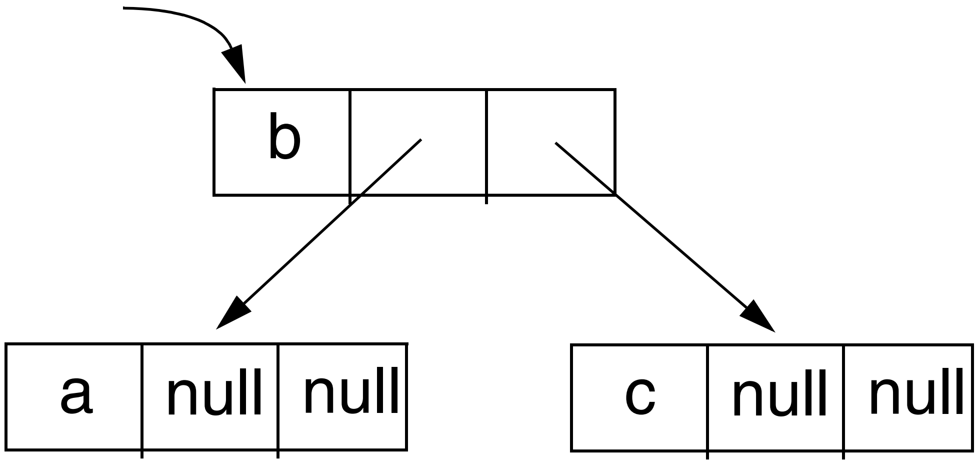
Subset

Representation

c) A linked list comprising the elements belonging to E:



d) A binary search tree, the elements of A being ordered:



Subset

Logic Properties

The logic properties of a subset are about the following ones:

1. It is possible to **insert** an element in a subset.
2. It is possible to **suppress** an element from a subset.
3. It is possible to know if an element **belongs** or not to a subset.
4. It is possible to know if a subset is **empty**.
5. It is possible to perform **set operations** on subsets: complement, union, intersection, difference and symmetric difference.
6. Some **axioms** must hold:

   Insert (x, E) => $x \in E$

   Suppress (x, E) => $x \notin E$

Logical Structure or Representation

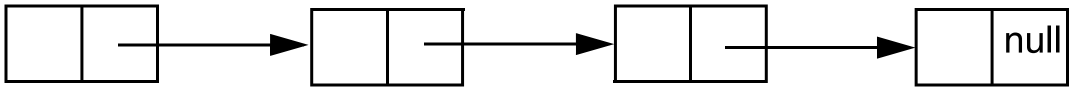
There are many sorts of lists: linear list, circular list, doubly-linked list, linear or circular, etc.

All kinds of data structures, like stacks and queues, can be implemented by lists.

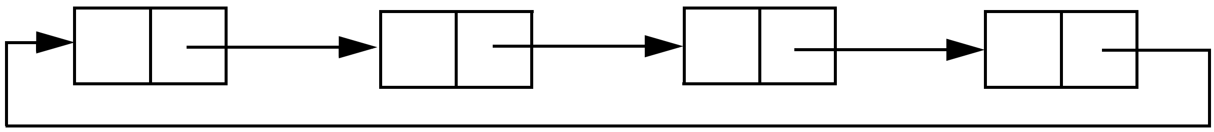
A list can therefore be a logical data structure (of low-level), or a representation structure.

Different Kinds of Lists

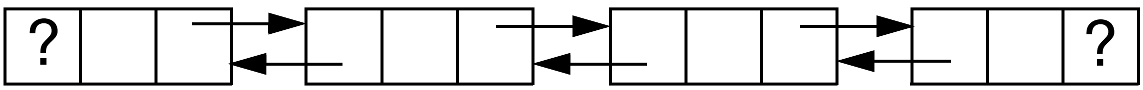
Linear list



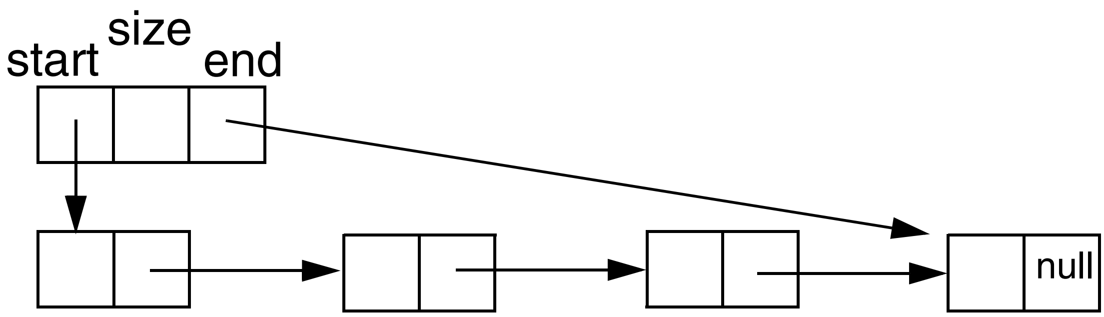
Circular list



Doubly-linked list

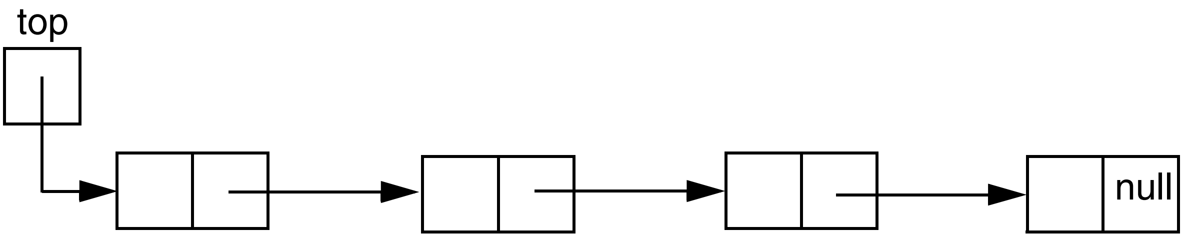


List with header



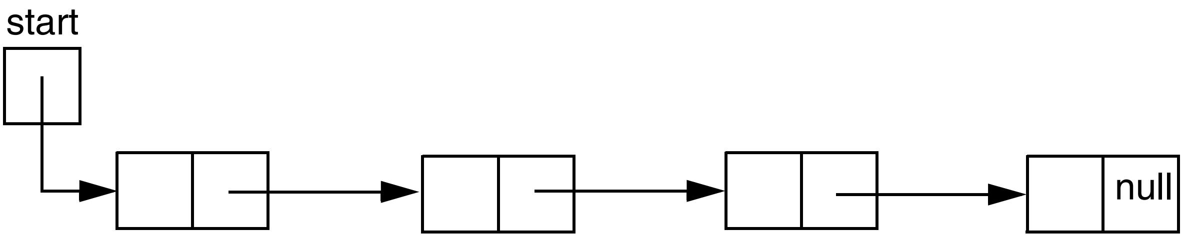
Representations by Lists

Stack



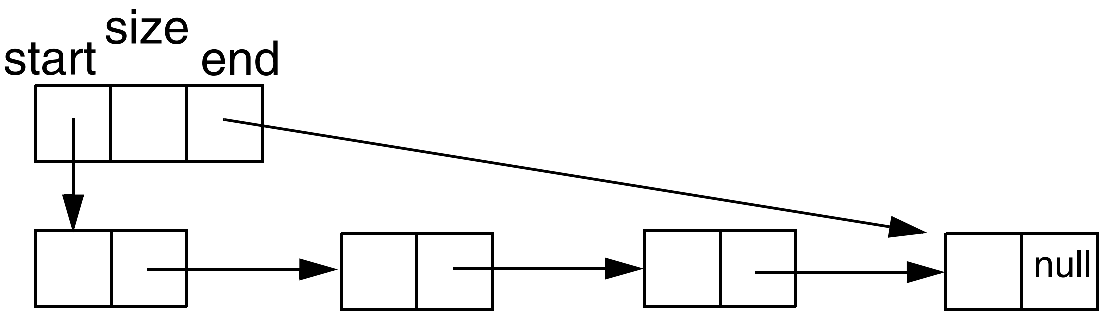
- Insertion and suppression in time O(1).

Queue with linear list



- Insertion in time O(1) and suppression in time O(n), or the contrary.

Queue with headed list



- One suppresses at the start, and inserts at the end. Both operations are therefore performed in time O(1).

Abstract Data Type

Definition

The representation of the data structure is hidden.

The only means for modifying the data structure or retrieving information about it is to call one of the operations associated with the abstract data type.

Abstract Data Type

Interface and Implementation

Abstract Data Type
=
Interface
+
Implementation

The interface defines the logical properties of the ADT, and especially the profiles or signatures of its operations.

The implementation defines the representation of the data structure and the algorithms that implement the operations.

Abstract Data Type

Realization in Ada

An ADT is realized by a package, most of the time a generic package.

The specification of the package is the interface of the ADT. The data structure is declared as a private type, or a limited private type. The subprograms having at least one parameter of the type are the operations of the ADT.

The private part of the specification and the body of the package provide the implementation of the ADT. The contain also the representation of the data structure.

A constant or variable of the ADT is called an object.

Abstract Data Type

Kinds of Operations

Constructors:

- Create, build, and initialize an object.

Selectors:

- Retrieve information about the state of an object.

Modifiers:

- Alter the state of an object.

Destructors:

- Destroy an object.

Iterators (parcoureurs, itérateurs):

- Access all parts of a composite object, and apply some action to each of these parts.

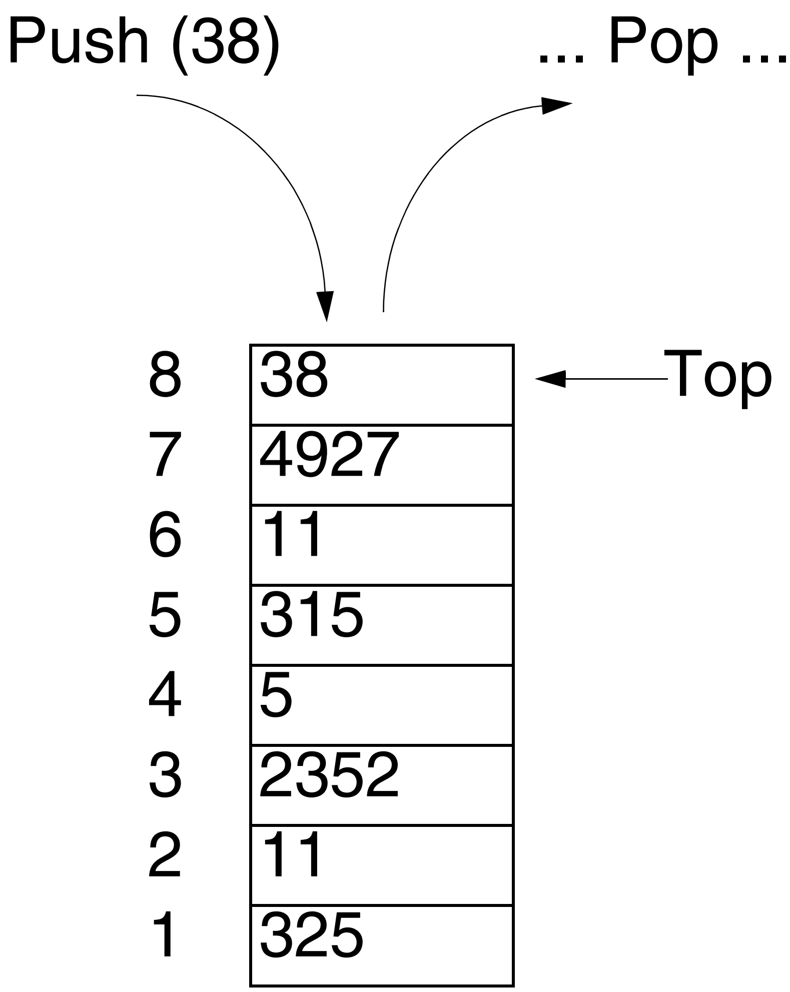
Abstract Data Type

Example: Set of Elements

```
Add (Set, Element)              -- constructor
Remove (Set, Element)           -- constructor
Iterate (Set, Action)           -- iterator
Is_A_Member (Set, Element)      -- selector
Make_Empty (Set)                -- constructor
Size (Set)                      -- selector
```

Abstract Data Type

Example: Stack



A stack is a "LIFO" list (last in, first out).

Abstract Data Type

Formal Definition of a Stack

E: is a set.

P: the set of stacks whose elements belong to E.

The empty set $\varnothing$ is a stack.

Operations

Push:	$P \times E$	$\rightarrow$	P	
Pop:	$P - \{\varnothing\}$	$\rightarrow$	P	(without access)
Top:	$P - \{\varnothing\}$	$\rightarrow$	E	(without removing)

Axioms

$\forall\, p \in P, \forall\, e \in E$:

$$\mathrm{Pop}\ (\mathrm{Push}\ (p, e)) = p$$

$$\mathrm{Top}\ (\mathrm{Push}\ (p, e)) = e$$

$\forall\, p \neq \varnothing$:

$$\mathrm{Push}\ (\mathrm{Pop}\ (p), \mathrm{Top}\ (p)) = p$$

Note: The axioms are necessary, because e.g. the operations on FIFO queues have exactly the same signatures!

Abstract Data Type

Primitive Operation

Note: Don't confuse with a primitive operation as defined by the Ada programming language.

First Definition

An operation is said to be primitive if it cannot be decomposed.

Example

- procedure Pop
  (S: in out Stack; E: out Element);

can be decomposed into:

- procedure Pop (S: in out Stack);
- function Top (S: Stack) return Element;

Abstract Data Type

Primitive Operation

Second Definition

An operation is said to be primitive if it cannot be implemented efficiently without access to the internal representation of the data structure.

Example

It is possible to compute the size of a stack by popping off all its element and then reconstructing it. Such an approach is highly inefficient.

Abstract Data Type

Sufficient Set of Operations

Definition

A set of primitive operations is sufficient if it covers the usual usages of the data structure.

Example

A stack with a Push operation but lacking a Pop operation is of limited value.

Is a stack without an iterator usable?

Abstract Data Type

Complete Set of Operations

Definition

A complete set of operations is a set of primitive operations including a sufficient set of operations and covering all possible usages of the data structure; otherwise stated, a complete set is a "reasonable" extension of a sufficient set of operations.

Example

Push, Pop, Top, Size and Iterate form a complete set of operations for a stack.

It would be possible to add Assign, "=", "/=" and Destroy.

Abstract Data Type

Stack: Specification in Ada

```
generic
   Max: Natural := 100;
   type Item_Type is private;
package Stack_Class_G is
   type Stack_Type is limited private;
   procedure Push (Stack: in out Stack_Type;
                   Item: in Item_Type);
   procedure Pop (Stack: in out Stack_Type);
   function Top (Stack: Stack_Type)
                  return Item_Type;
   generic
      with procedure Action
                  (Item: in out Item_Type);
   procedure Iterate (Stack: in Stack_Type);
   Empty_Error: exception;
   -- raised when an item is accessed or popped from an empty stack.
   Full_Error: exception;
   -- raised when an item is pushed on a full stack.
```

Abstract Data Type

Stack: Specification in Ada

```
private
    type Table_Type is array (1..Max)
            of Item_Type;
    type Stack_Type is record
        Table: Table_Type;
        Top: Integer range 0..Max := 0;
    end record
end Stack_Class_G;
```

Abstract Data Type

Stack: Specification in Ada

Unfortunately, the interface does not show only logical properties. The implementation slightly shows through, by the generic parameter Max and the exception Full_Error, for instance.

The exception Empty_Error is added in order to extend the domains (of definition/validity) of the operations Pop and Top.

IV. Trees

Kinds of trees
Binary tree
Traversal of a binary tree
Search tree
Expression tree
Polish forms
Strictly binary tree
Almost complete binary tree
Heap

Kinds of Trees

Tree (Arbre)

- Ordered Tree (Arbre ordonné)
  - (children are ordered)
- 2-Tree (Arbre d'ordre 2)
  - (every node has 0 or 2 children)
- Trie (from re<u>trie</u>val)
  - (also called "Lexicographic Search Tree")
  - (a trie of order m is empty or is a sequence of m tries)

Binary Tree (Arbre binaire)

- Binary Search Tree (Arbre de recherche)
- AVL-Tree (Arbre équilibré)
- Heap (Tas)

Multiway Search Tree

- B-Tree

Binary Tree

A binary tree is a finite set E, that is empty, or contains an element r and whose other elements are partitioned in two binary trees, called left and right subtrees.

r is called the root (racine) of the tree. The elements are called the nodes of the tree.

A node without a successor (a tree whose left and right subtrees are empty) is called a leaf.

Binary Tree

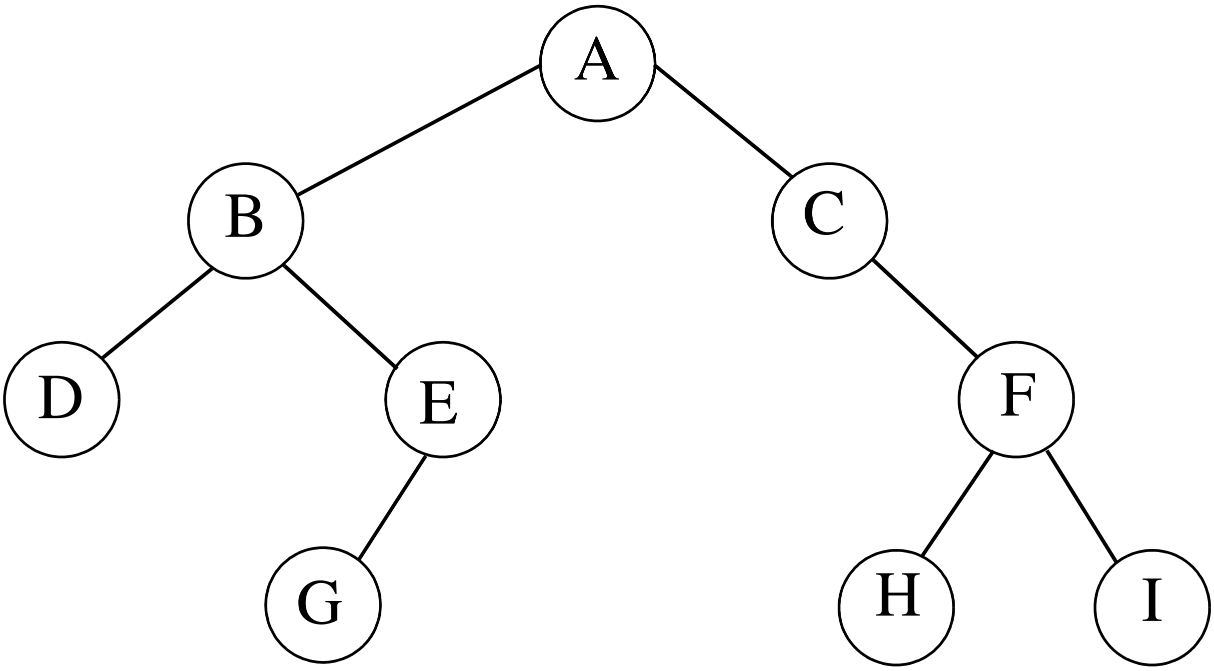
E is a finite set

(i) E is empty

or

(ii) $\exists\ r \in E, \exists\ E_g, \exists\ E_d,$
$r \notin E_g, r \notin E_d,$
$E_g \cap E_d = \varnothing, E = \{r\} \cup E_g \cup E_d$

Binary Tree

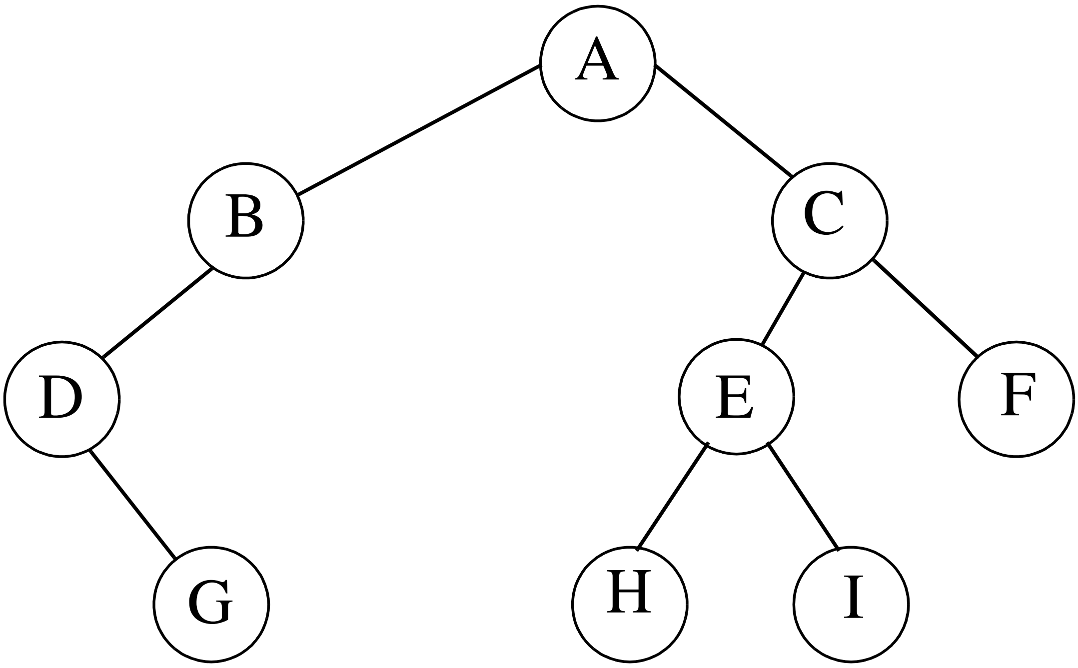


The two examples at the bottom are distinct binary trees, but identical trees.

Traversal of a Binary Tree

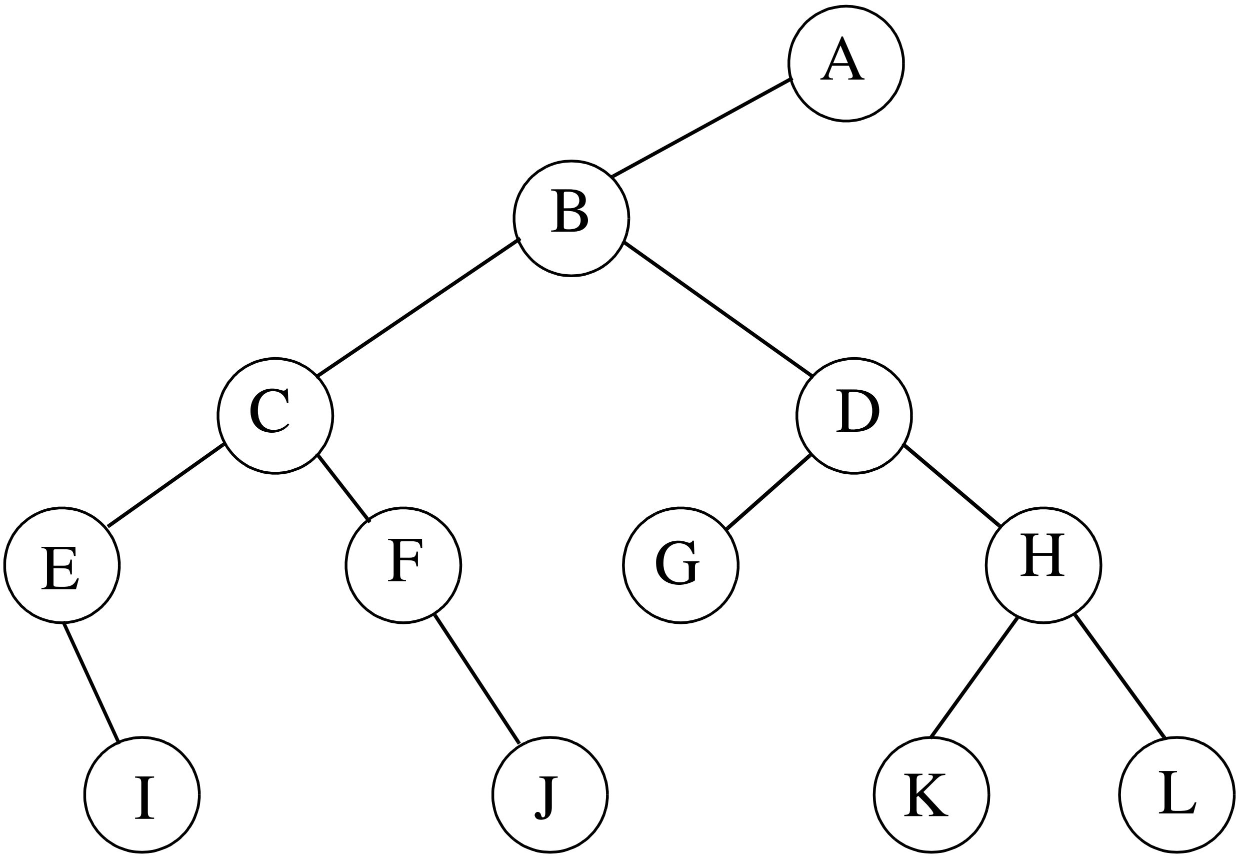
1. Preorder or depth-first order
   (préordre ou en profondeur d'abord)
   (i) visit the root
   (ii) traverse the left subtree
   (iii) traverse the right subtree
2. Inorder or symmetric order
   (inordre ou ordre symétrique)
   (i) traverse the left subtree
   (ii) visit the root
   (iii) traverse the right subtree
3. Postorder
   (postordre)
   (i) traverse the left subtree
   (ii) traverse the right subtree
   (iii) visit the root
4. Level-order or breadth-first order
   (par niveau)
   Visit all the nodes at the same level, starting with level 0

Traversal of a Binary Tree



Preorder:	A B D G C E H I F
Inorder:	D G B A H E I C F
Postorder:	G D B H I E F C A
By level:	A B C D E F G H I

Traversal of a Binary Tree



Preorder:

Inorder:

Postorder:

By level:

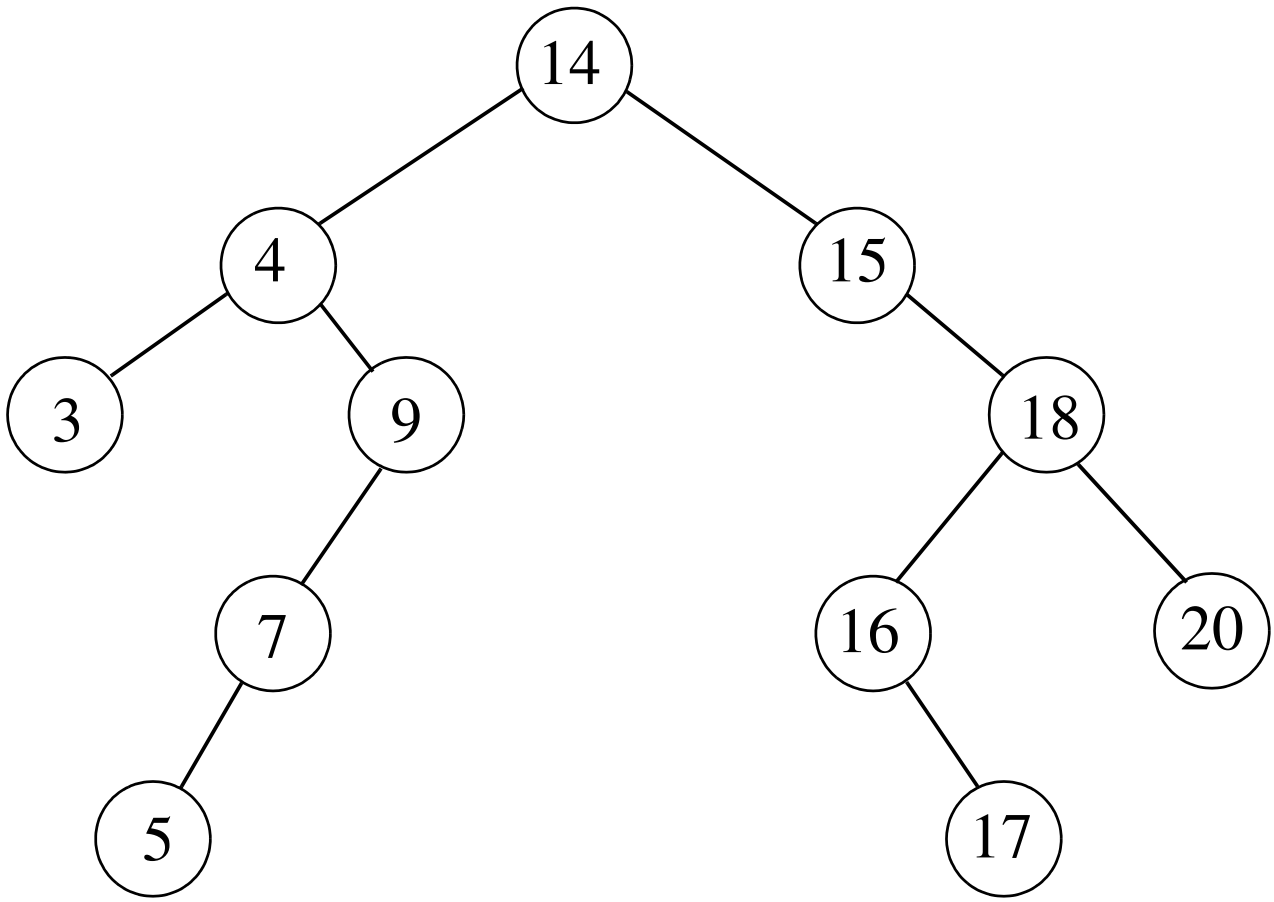
Search Tree

A search tree is a special case of a binary tree.

Each node contains a key and the following relationship is satisfied for each node:

$\forall n, \forall n_1 \in E_g (n), \forall\ n_2 \in E_d (n)$
$key\ (n_1) \leq key\ (n) \leq key\ (n_2)$

Search Tree



Inorder: 3 4 5 7 9 14 15 16 17 18 20

Application: Sorting

Input: 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17

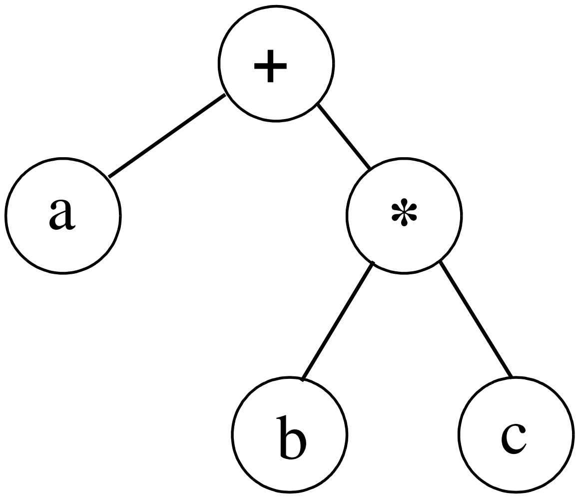
Processing: Build the tree

Result: Traverse in inorder

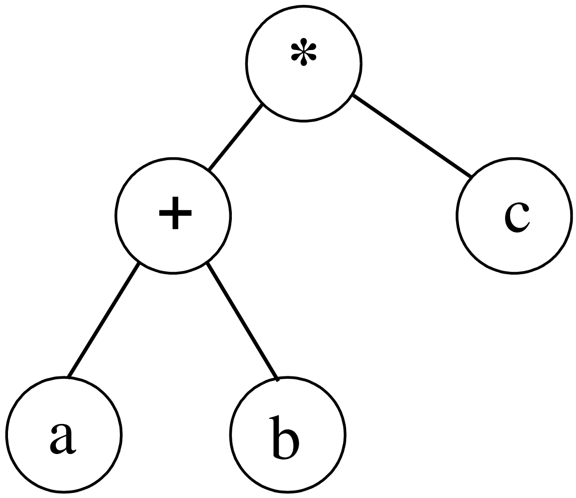
Application: Searching

Expression Tree

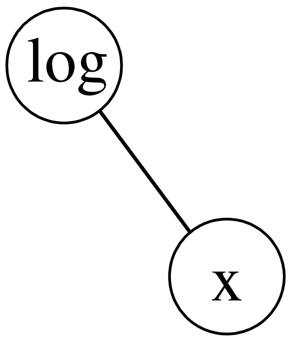
An expression tree is a binary tree whose leaves contain values (numbers, letters, variables, etc.) and the other nodes contain operation symbols (operations to be performed on such values).



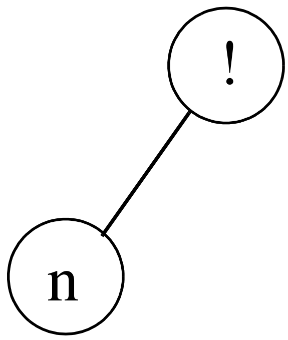
(i) a+b*c



(ii) (a+b)*c

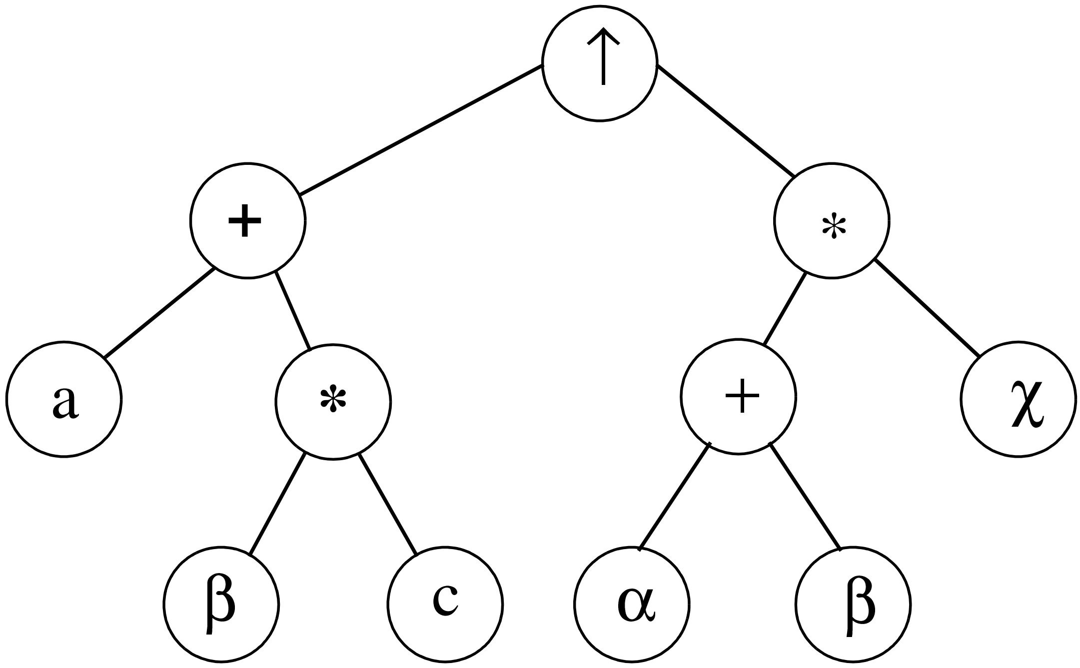


(iii) log x



(iv) n!

Expression Tree



Polish Forms (Notations polonaises)

(i) Prefix form (Notation préfixée)

The operator is written **before** the operands

$\rightarrow$ preorder

$\uparrow + a * b\ c * + a\ b\ c$

(ii) Infix form (Notation infixée ou symétrique)

The operator is written **between** the operands

$\rightarrow$ inorder

$a + b * c \uparrow (a + b) * c$

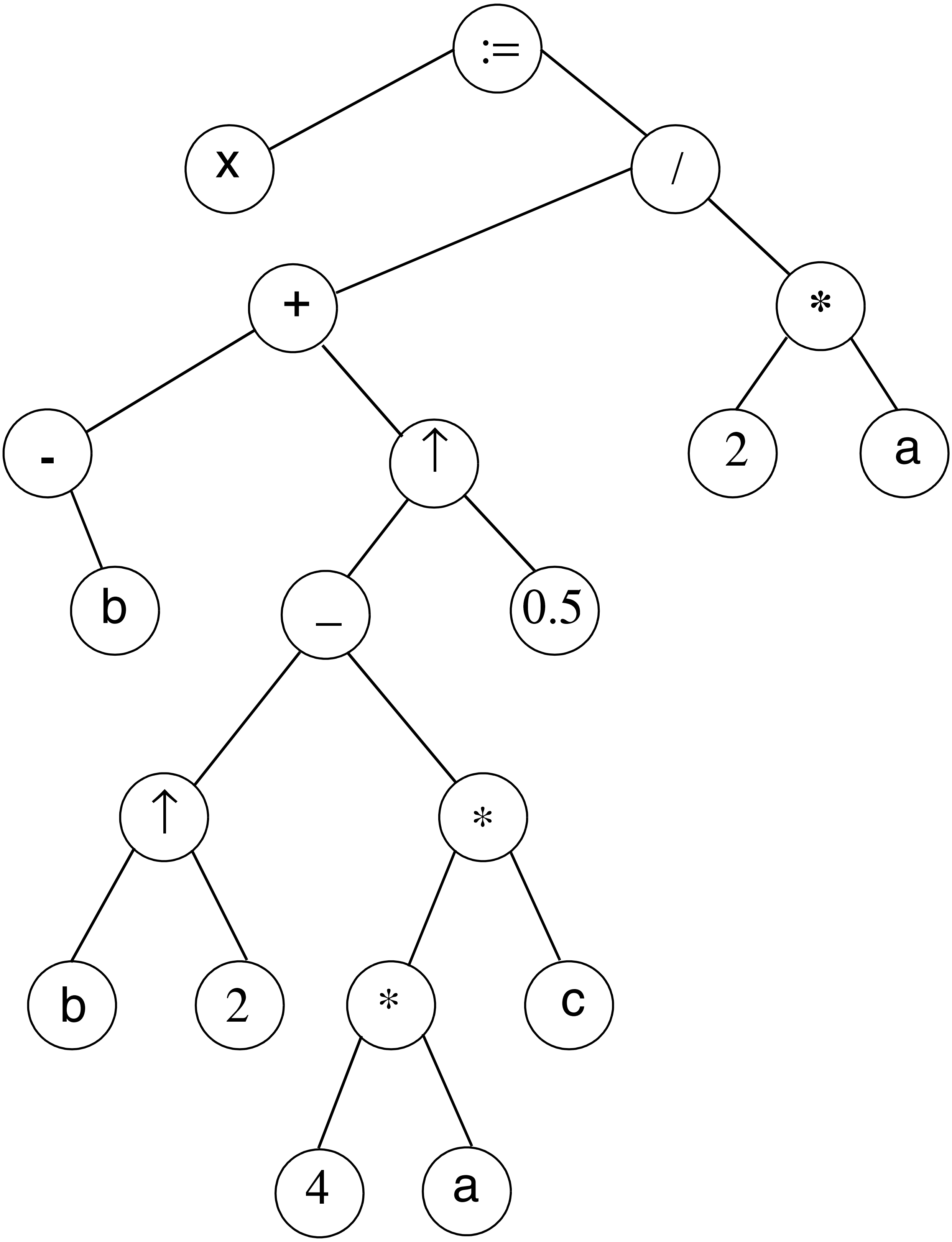
(iii) Postfix form (Notation postfixée)

The operator is written **after** the operands

$\rightarrow$ postorder

$a\ b\ c * + a\ b + c * \uparrow$

Expression Tree



Other Trees

Strictly Binary Tree (Arbre strictement binaire)

Any node that is not a leaf has non empty left and right subtrees.

Almost Complete Binary Tree (Arbre binaire presque complet)

(i) Each leaf of the tree is at the level k or k+1;

(ii) If a node in the tree has a right descendant at the level k+1, then all its left descendants that are leaves are also at the level k+1.

Heap (Tas)

(i) A heap is an almost complete binary tree.

(ii) The contents of a node is always smaller or equal to that of the parent node.

V. Graphs

Definitions

Oriented Graph (example and definitions)

Undirected Graph (example and definitions)

Representations

- Adjacency Matrix
- Adjacency Sets
- Linked Lists
- Contiguous Lists (matrices)
- "Combination"

Abstract Data Type

List of Algorithms

Traversal

Shortest path

Representation of a weighted graph

Dijkstra’s Algorithm

Principle of dynamic programming

Graphs

Definitions

1. **Directed graph, digraph** (graphe orienté):

- $G = (V, E)$
- V finite set of vertices (sommet)
- $E \subset V \times V$ set of arcs (arc)

This definition prohibits multiple parallel arcs, but self-loops (v, v) are allowed.

2. **Undirected graph, graph** (graphe non orienté)

- $G = (V, E)$
- V finite set of nodes (noeud)
- E a set of two-element subsets of V, $\{\{y, z\} | x, y, z \in V\}$, set of edges (arête).

This definition prohibits self loops like $\{v\}$.

Graphs

Definitions

3. **Weighted (directed) graph** (graphe valué)
A value is associated with each arc or edge, often an integral number, sometimes the value is composite, i.e. is a tuple.

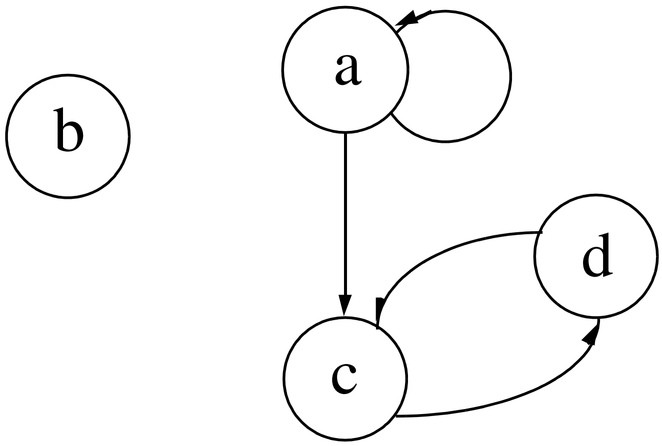
4. A **network** (réseau) is a weighted directed graph.
The values might represent distances, transportation capacities, bandwidth, throughput, etc.

The complexity of graph algorithms are usually measured as functions of the number of vertices and arcs (nodes and edges).

Sometimes the terms "node" and "edge" are also used for digraphs. Sometimes "vertex" is used instead of edge for undirected graphs.

Directed Graphs

Example



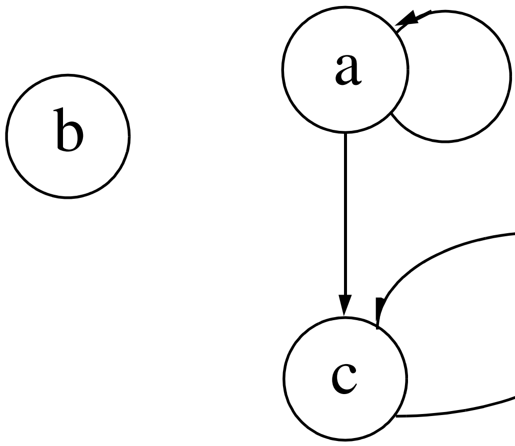
V = {a, b, c, d}

E = {(a, a), (a, c), (c, d), (d, c)}

- (a, a) is a self-loop (boucle)
- multiple parallel arcs are prohibited (E is a set!)

Directed Graphs

Example



1.1. a is a predecessor (prédecesseur) of c and c is a successor (successeur) of a.

1.2. The indegrees (degrés incidents à l'intérieur) are:

0 for b, 1 for a, 2 for c, 1 for d.

The outdegrees (degrés incidents à l'extérieur) are:

0 for b, 2 for a, 1 for c, 1 for d.

1.3. (a, c, d, c) is a path (chemin).

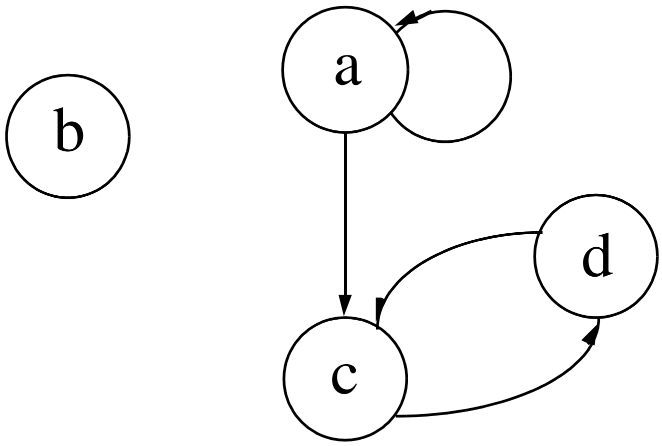
1.4. (c, d, c, d, c) is a cycle (circuit).

1.5. (a, c, d) is a simple path (chemin simple).

(c, d, c) et (d, c, d) are simple cycles (circuits simples).

Directed Graphs

Example



1.6. c and d are strongly connected (fortement connexes).

The digraph itself is not strongly connected.

1.7. ({a, c, d}, {(a, c), (c, d), (d, c)}) is a subgraph (sous-graphe (partiel)).

1.8. and 1.9.

The digraph does not have a spanning tree (arbre de sustension).

The subgraph:

({a, c, d}, {(a, a), (a, c), (c, d), (d, c)})

has as a spanning tree:

({a, c, d}, {(a, c), (c, d)})

Directed Graphs

Definitions

1.1. If $(v, w) \in E$ then v is a **predecessor** (prédécesseur) of w, and w is a **successor** (successeur) of v.

1.2 The **outdegree** ((demi-)degré incident vers l'extérieur) of a vertex is its number of successors.

The **indegree** ((demi-)degré incident vers l'intérieur) of a vertex is its number of predecessors.

1.3. An (oriented) **path** (chemin (orienté)) is a sequence $(v_1, v_2, \ldots, v_k)$ of V such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$.

1.4. A path $(v_1, v_2, \ldots, v_k)$ such that $v_1 = v_k$ is a **cycle** (circuit).

1.5. If the vertices of a path are all distinct, expect the first and last one, then the path is said to be **simple** (chemin simple).

Directed Graphs

Definitions

1.6. Two vertices are **strongly connected** (fortement connexes) if there are paths connecting each one to the other.

A digraph is strongly connected if all its vertices are strongly connected.

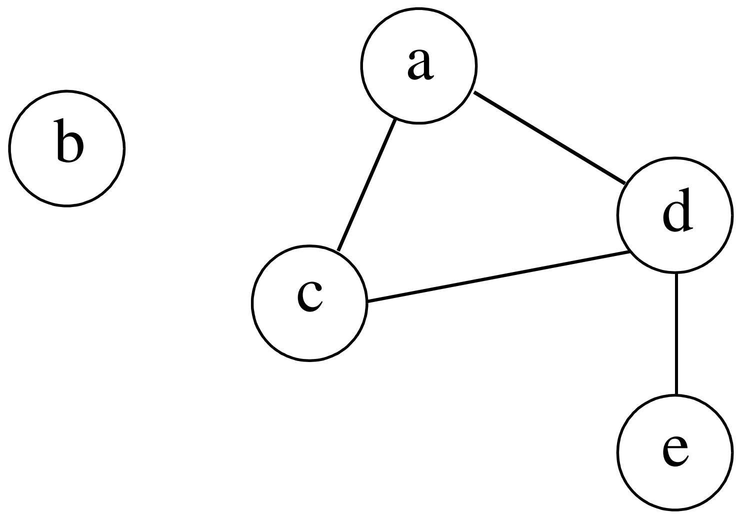
1.7. A **subgraph** (sous-graphe (partiel)) is a digraph (V', E') such that $V' \subset V$ and $E' \subset E$.

1.8. A (rooted) **tree** (arbre) is a digraph having a vertex, called its **root** (racine), having the property: For each vertex of the graph there is exactly one path from the root to the vertex.

1.9. A **spanning tree** (arbre de sustension) of a digraph (V, E) is a subgraph T = (V', E') that is a tree and such that V = V'.

Undirected Graphs

Example



V = {a, b, c, d, e}

E = {{a, c}, {a, d}, {c, d}, {d, e}}

- self-loops (boucle) are prohibited.
- multiple parallel edges are prohibited (E is a set!).

Undirected Graphs

Definitions

1.1. If $(v, w) \in E$, then the nodes v and w are said to be **adjacent** (voisins, adjacents).

1.2. The **degree** (degré) of a node is the number of its adjacent nodes.

1.3. A sequence of nodes $(v_1, v_2,...,v_k)$ of V such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ is a **path** (chaîne).

1.4. A path $(v_1, v_2,...,v_k)$ such that $v_1 = v_k$ is a **cycle** (cycle).

1.5. If all nodes are distinct, the path or cycle is said to be **simple** (chaîne simple, cycle simple).

Undirected Graphs

Definitions

1.6 Two nodes are **connected** (connexe) if there is a path going from one to the other.

The graph is said to be connected if all its nodes are connected.

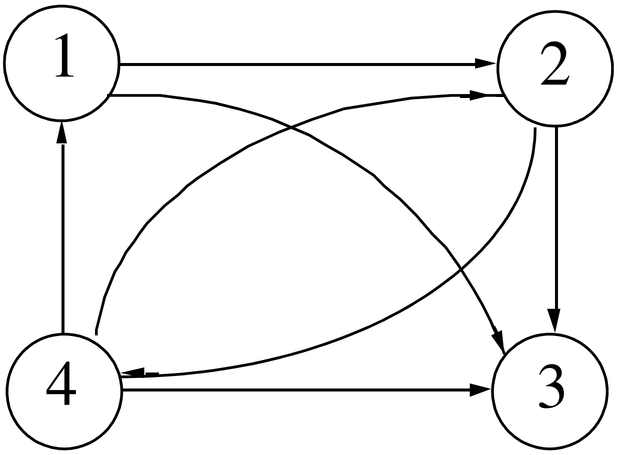
1.7 A **subgraph** (sous-graphe (partiel)) is a graph (V', E') such that V' $\subset$ V and E' $\subset$ E.

1.8 A a **tree** or free tree (arborescence) is a graph where there is exactly one simple path between every pair of nodes.

1.9. A **spanning tree** (arbre de sustension) of a graph (V, E) is a subgraph T = (V', E') that is a tree and such that V = V'.

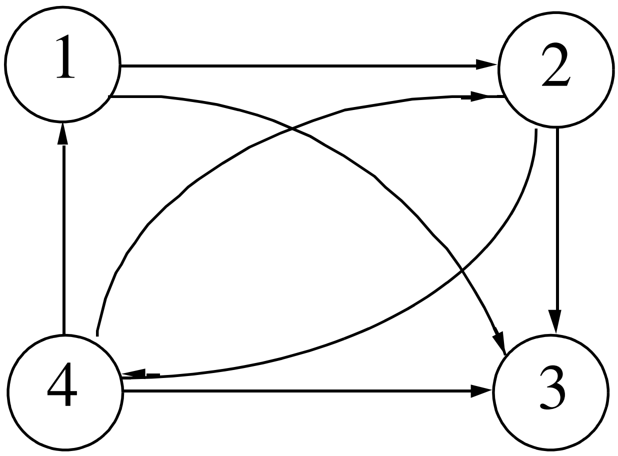
Graphs

Representations



- Adjacency matrix
- Adjacency sets (or lists)
- Linked lists
- Contiguous lists (matrices)
- "Combinations"

Adjacency Matrix



	1	2	3	4
1		T	T	
2			T	T
3				
4	T	T	T	

a (i, j) = T

<=>

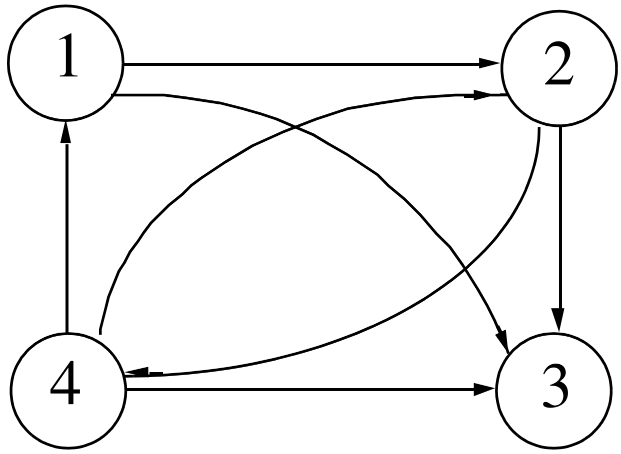
(i, j) is an arc

T stands for true, i.e. there is an arc.
Empty cells have value F.

Adjacency Matrix

```
subtype Nb_of_Vertices is Natural range 0..Max;
subtype Vertex_Type is Positive range 1..Max;

type Matrix_Type is
    array (Vertex_Type range <>,
              Vertex_Type range <>) of Boolean;

type Graph_Type (Size: Nb_of_Vertices := 0) is record
    Adjaceny_Matrix: Matrix_Type (1..Size, 1..Size);
end record;
```

Adjacency Sets



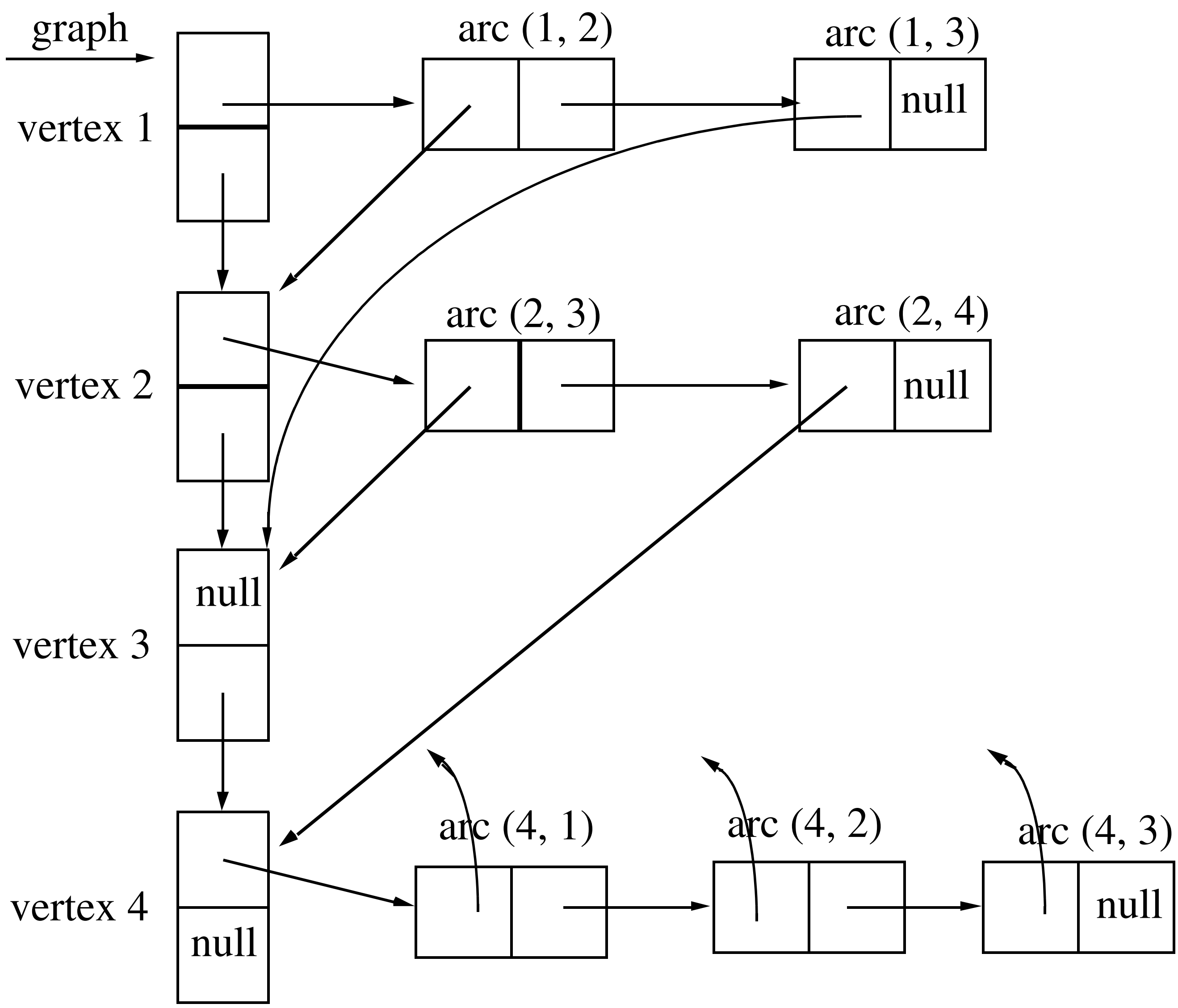
1	2, 3
2	3, 4
3	
4	1, 2, 3

$\{(1, \{2, 3\}), (2, \{3, 4\}), (3, \emptyset), (4, \{1, 2, 3\})\}$

Adjacency Sets

```
subtype Nb_of_Vertices is Natural range 0..Max;
subtype Vertex_Type is Positive range 1..Max;

package Set is new Set_G (Element_Type => Vertex_Type);
type Set_of_Vertices is new Set.Set_Type;

type Adjacency_Set:Type is
    array (Vertex_Type range <>)
        of Set_of_Vertices;

type Graph_Type (Size: Nb_of_Vertices := 0) is record
    Adjacency_Sets: Adjacency_Set_Type (1..Size);
end record;
```

Linked Lists



It would be possible to add additional links:

- from each arc to its starting vertex;
- from each vertex, link together all the arcs of which it is the final vertex.

Linked Lists

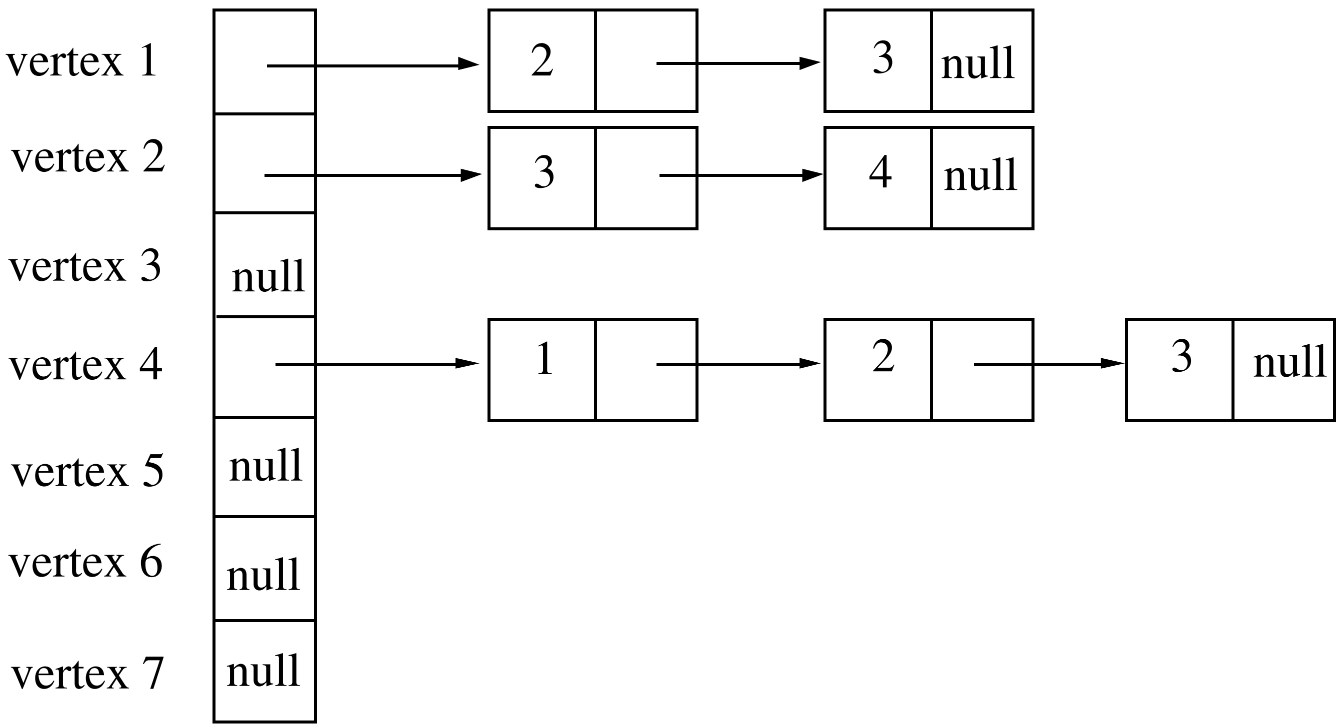
```
type Vertex_Type;
type Edge_Type;
type Vertex_Access_Type is
    access Vertex_Type;
type Edge_Access_Type is
    access Edge_Type;

type Vertex_Type is record
    First_Edge: Edge_Access_Type;
    Next_Vertex: Vertex_Access_Type;
end record;

type Edge_Type is record
    End_Vertex: Vertex_Access_Type;
    Next_Edge: Edge_Access_Type;
end record;

type Graph_Type is
    new Vertex_Access_Type;
```

Contiguous Lists (Matrices)

Vertex	Number	List
1	2	2 3 - - - - -
2	2	3 4 - - - - -
3	0	- - - - - - -
4	3	1 2 3 - - - -
5	-	- - - - - - -
6	-	- - - - - - -
max = 7	-	- - - - - - -

For each vertex, the vertices it is connected to by an arc are listed. The number of such vertices equals at most the number of vertices, and an n x n matrix is hence sufficient.

"Combination"



Because the "vector" of vertices has length 7 in the example, at most 7 vertices are possible.

Graphs

Abstract Data Type

```
generic
   type Vertex_Value_Type is private;
   type Edge_Value_Type is private;
package Graph_G is
   type Graph_Type is limited private;
   type Vertex_Type is private;
   type Edge_Type is private;

   -- operations to set and consult the values of vertices and edges.
   procedure Set
      (Vertex: in out Vertex_Type;
      Value: in Vertex_Value_Type);
   function Value
      (Vertex: Vertex_Type)
      return Vertex_Value_Type;

   -- similar for edges
   …
```

Graphs

Abstract Data Type

```
procedure Add
   (Vertex: in out Vertex_Type;
   Graph: in out Graph_Type);
procedure Remove
   (Vertex: in out Vertex_Type;
   Graph: in out Graph_Type);
procedure Add
   (Edge: in out Edge_Type;
   Graph: in out Graph_Type;
   Source,
   Destination: in Vertex_Type);
procedure Remove
   (Edge: in out Edge_Type;
   Graph: in out Graph_Type);
```

Graphs

Abstract Data Type

```
function Is_Empty
   (Graph: Graph_Type)
   return Boolean;
function Number_of_Vertices
   (Graph: Graph_Type)
   return Natural;
function Source
   (Edge: Edge_Type)
   return Vertex_Type;
function Destination
   (Edge: Edge_Type)
   return Vertex_Type;
```

Graphs

Abstract Data Type

```
generic
   with procedure Process
      (Vertex: in Vertex_Type;
      Continue: in out Boolean);
procedure Visit_Vertices
   (Graph: in Graph_Type);

generic
   with procedure Process
      (Edge: in Edge_Type;
      Continue: in out Boolean);
procedure Visit_Edges
   (Graph: in Graph_Type);
```

Graphs

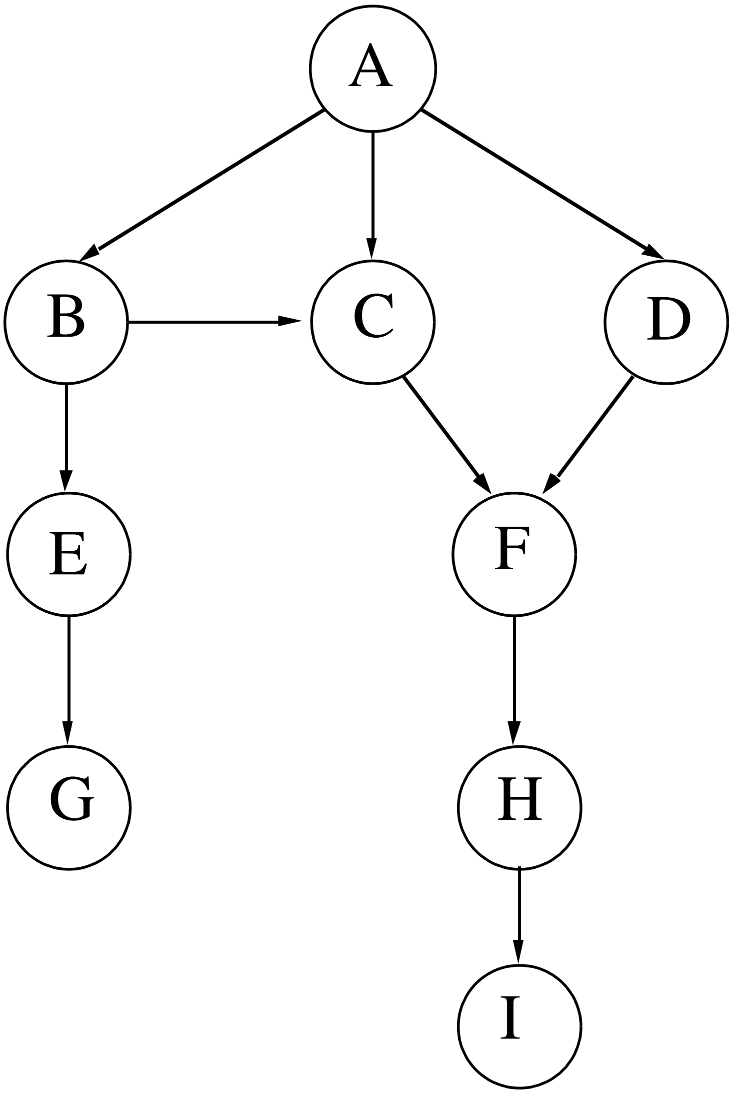
Abstract Data Type

```
    generic
        with procedure Process
            (Edge: in Edge_Type;
            Continue: in out Boolean);
    procedure Visit_Adj_Edges
        (Vertex: in Vertex_Type
        [; Graph: in Graph_Type]);
    ...
end Graph_G;
```

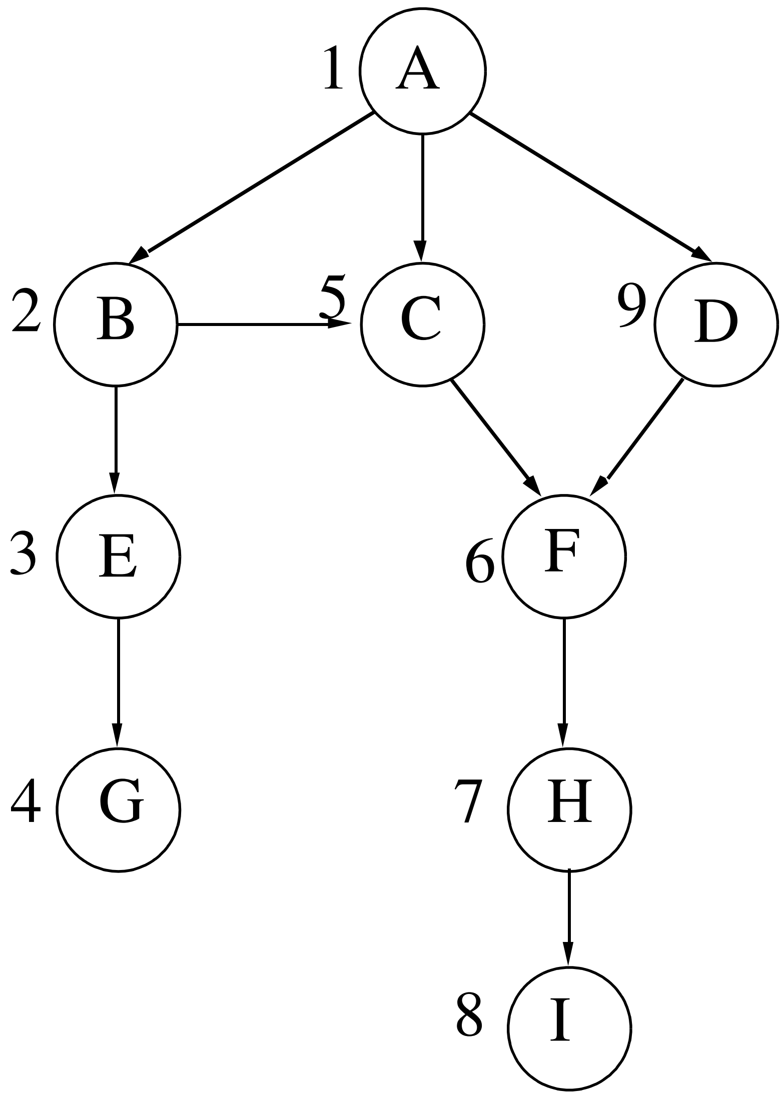
Graph Algorithms

Depth-first search
Breadth-first search
Connectivity problems
Minimum Spanning Trees
Path-finding problems
Shortest path
Topological sorting
Transitive Closure
The Newtwork Flow problem
    (Ford-Fulkerson)
Matching
    Stable marriage problem
Travelling Salesperson problem
Planarity problem
Graph isomorphism problem

Graph Traversal

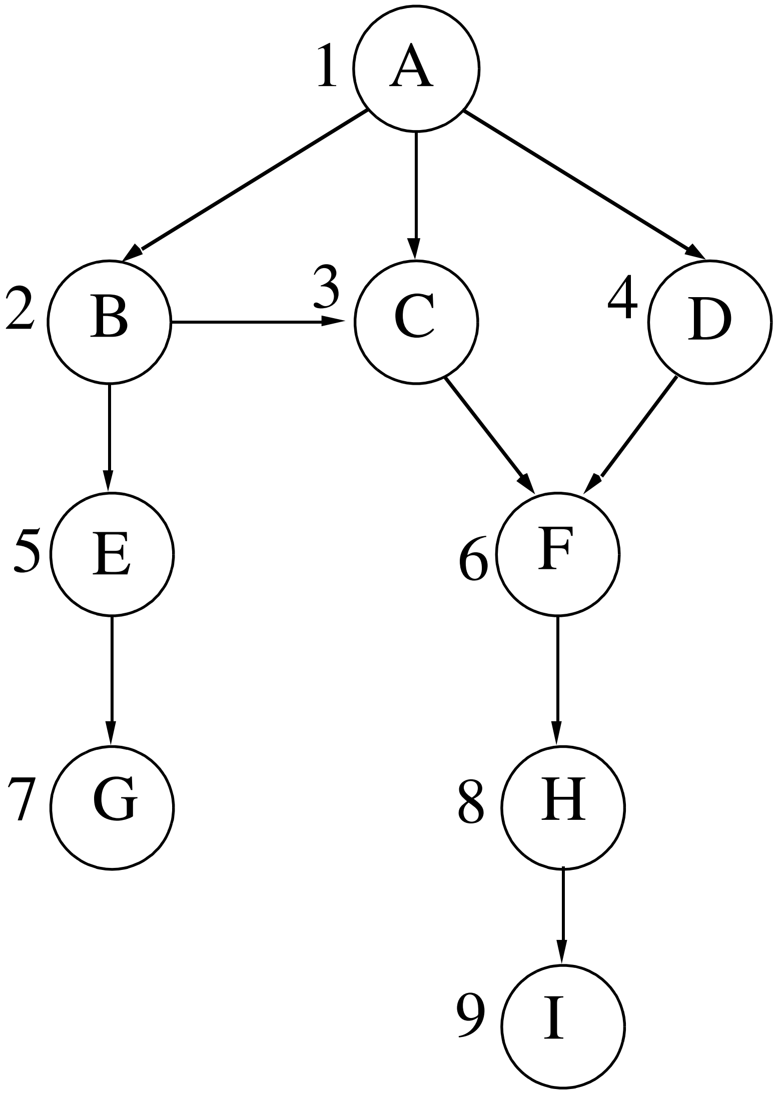


Depth-First Search



(A, B, E, G, C, F, H, I, D)

Breadth-First Search



(A, B, C, D, E, F, G, H, I)

Depth-First Search

For each vertex v in the graph:

1. visit the vertex v;
2. determine the vertices adjacent to v:
   w_1, w_2,...w_k;
3. for i varying from 1 to k: traverse starting from vertex w_k.

Don't forget to mark the vertices already visited.

Depth-First Search

```
-- pseudo-Ada

generic
    with procedure Visit (Vertex: in Vertex_Type);
procedure Depth_First (Graph: in Graph_Type);

procedure Depth_First (Graph: in Graph_Type) is
    Visited: array (Graph.Vertex_Set) of Boolean;
    procedure Traverse (Vertex: Vertex_Type) is separate;
begin
    for all Vertex in Graph.Vertex_Set loop
        Visited (Vertex) := False;
    end loop;
    for all Vertex in Graph.Vertex_Set loop
        if not Visited (Vertex) then
            Traverse (Vertex);
        end if;
    end loop;
end Depth_First;
```

Depth-First Search

```
separate (Depth_First)
procedure Traverse (Vertex: in Vertex_Type) is
begin
    Visited (Vertex) := True;
    Visit (Vertex);
    for all W adjacent to Vertex loop
        if not Visited (W) then
            Traverse (W);
        end if;
    end loop;
end Traverse;
```

Breadth-First Search

For each vertex v in the graph:

1. visit the vertex v;
2. visit the vertices adjacent to v:
   w_1, w_2,...w_k;
3. then visit the vertices adjacent to w_1, then those adjacent to w_2, etc.

Don't forget to mark the vertices already visited.

Breadth-First Search

```
package Queue is
   new Queue_G
      (Element_Type => Vertex_Type);

type Queue_of_Vertices is
   new Queue.Queue_Type;

generic
   with procedure Visit
      (Vertex: in Vertex_Type);
procedure Breadth_First
   (Graph: in Graph_Type);
```

Breadth-First Search

```
procedure Breadth_First (Graph: in Graph_Type) is
    Visited: array (Graph.Vertex_Set) of Boolean
                := (others => False);
    Waiting: Queue_of_Vertices;
    Next: Vertex_Type;
begin
    for all Vertex in Graph.Vertex_Set loop
        if not Visited (Vertex) then
            Insert (Waiting, Vertex);
            while not Is_Empty (Waiting) loop
                Remove (Waiting, Next);
                Visited (Next) := True;
                Visit (Next);
                for all W adjacent to Next loop
                    if not Visited (W) then
                        Insert (Waiting, W);
                    end if;
                end loop;
            end loop;
        end if;
    end loop;
end Breadth_First;
```

Shortest Path

The graph is weighted: a positive numeric value is associated with each arc.

Statement 1:

Given a vertex Start and a vertex Target, find the shortest path from Start to Target.

Statement 2:

Given a vertex Start, find the shortest paths from Start to all other vertices.

- Dijkstra's Algorithm (especially when adjacency lists are used for the representation)
- Floyd's Algorithm (especially when an adjacency matrix is used for the representation)

Representation of a Weighted Graph

The function Weight is defined for all couples of vertices:

Weight (V, V) = 0

Weight (V, W) =

$\propto$ (infinity) if there is no arc from V to W;

the value of the arc, if there is one;

Weight can be implemented by a matrix or another representation, e.g. a map or dictionary.

Dijkstra's Algorithm

Principle

Start: starting vertex

S: Set of vertices for which the length of the shortest path is known.

Q: Set of vertices adjacent to S.

d (V): Distance between Start and V, for $V \in S \cup Q$, with the meaning:

- If $V \in S$, it is the length of the shortest path;
- If $V \in Q$, it is the length of the shortest path via S (all vertices on the path are in S, except V itself).

Dijkstra’s Algorithm

Principle

1. Initialization

Q := {Start} d(Start) := 0;

S := Ø

2. Loop

2.1. Extract from Q the vertex C having the smallest distance:

$$d\ (C) = \min\ (d\ (V);\ V \in Q)$$

2.2. Add C to S (see Justification)

2.3. Add the vertices adjacent to C to Q, and update their distances:

For every W adjacent to C:

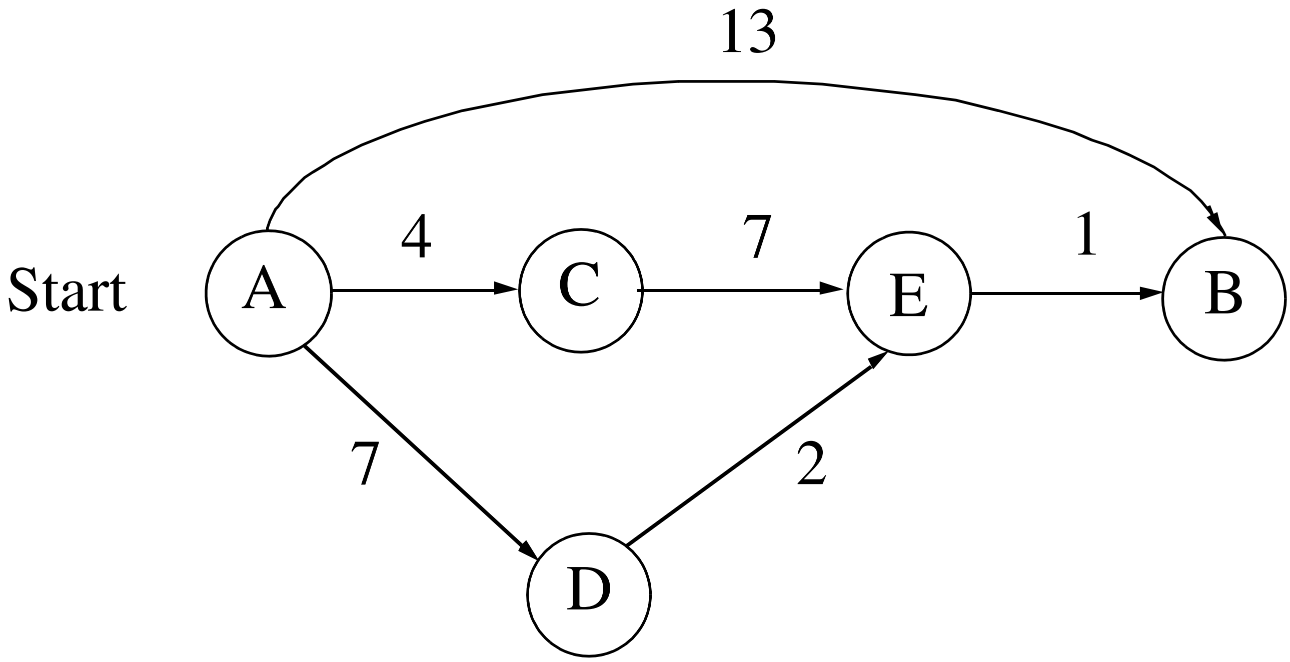
- if $W \notin Q$: d(w) := d(C) + weight (C, W)
- if $W \in Q$: d(w) :=
  min (d(W),d(C) + weight (C, W))

3. Stop condition

- Q is empty

Dijkstra's Algorithm

Example



Dijkstra's Algorithm

Example

Initialization

Q := {A}, S := ∅, d (A) := 0

First Loop (process A)

S := {A}, Q := {B, C, D}

d (B) := 13, d (C) := 4, d (D) := 7

Second Loop (process C)

S := {A, C}, Q := {B, D, E}

d (B) = 13, d(D) = 7, d (E) := 11

because d (E) := d (C) + weight (C, E)

Third Loop (process D)

S := {A, C, D}, Q := {B, E}

d (B) = 13, d (E) := 9, because

d (E) :=

min (previous value,d(D) + weight (D, E))

Fourth Loop (process E)

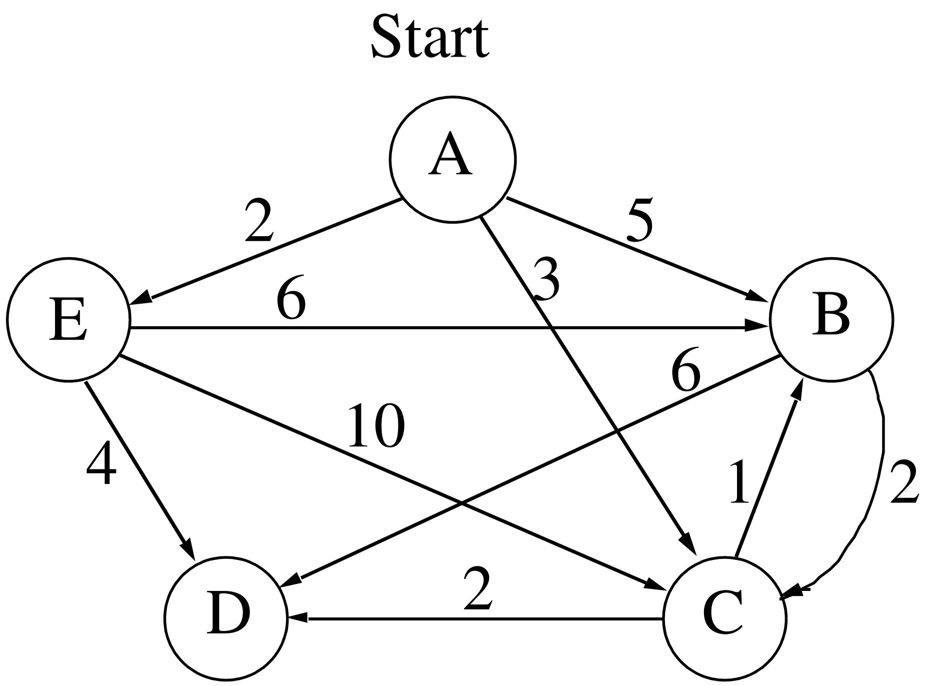
S := {A, C, D, E}, Q := {B}

d (B) := 10

Fifth and Last Loop (process B)

S := {A, B, C, D, E}, Q := ∅

Other Example

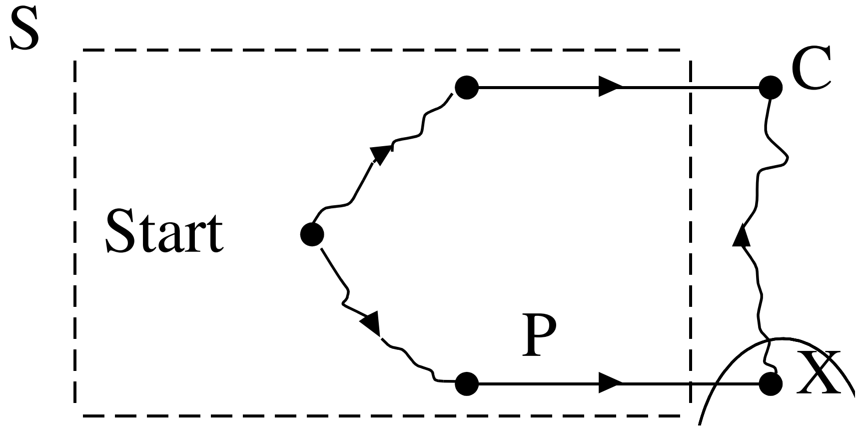


Dijkstra's Algorithm

Justification

Suppose there is a shorter path P going to C.

Then P necessarily goes through a vertex not belonging to S. Let X be the first vertex on P which is not in S:



Since X is adjacent to S, X belongs to Q and d (X) is the length of the shortest path via S.

But by the very choice of C: d (X) $\geq$ d (C) and the length of P is necessarily greater or equal to d (X).

Dijkstra's Algorithm

Implementation using a Priority Queue

Precondition:
Weight (V, W) = ∝ if there is no arc from V to W.

```
Q: Priority_Queue_Type;
C: Vertex_Type;

Distance := (others => ∝);
Insert (Q, Start);
Distance (Start) := 0;

while not Is_Empty (Q) loop
    Remove (Q, C);
    for all W adjacent to C loop
        if Distance (C) + Weight (C, W) < Distance (W) then
            Distance (W) := Distance (C) + Weight (C, W);
            Insert (Q, W);
        end if;
    end loop;
end loop;
```

Dijkstra's Algorithm

Implementation with a Set

Precondition:

Weight (V, W) = $\propto$ if there is no arc between V and W; and Weight (V, W) = 0 if V = W.

```
S: Set_of_Vertices;
Start, C: Vertex_Type;
Min_Dist: Weight_Type;
Found: Boolean;

Insert (S, Start);
for all V in Graph.Vertex_Set loop
   Distance (V) := Weight (Start, V);
end loop;
```

Dijkstra's Algorithm

Implementation with a Set

```
Found := True;
while Found loop
    -- at each pass, en element is added to S
    Found := False;
    Min_Dist = ∝;
    -- Find the element to be added to S
    for all V in Graph.Vertex_Set loop
        if V not in S then
            if Distance (V) < Min_Dist then
                Found := True;
                Min_Dist := Distance (V);
                C := V;
            end if;
        end if;
    end loop;
    if Found then
        Insert (S, C);
        for all W adjacent to C loop
            if Min_Dist + Weight(C,W) < Distance(W) then
                Distance(W) := Min_Dist + Weight(C,W);
            end if;
        end loop;
    end if;
end loop;
```

Find the paths rather than their lengths

Representation of a path

- For each vertex on the path, store its predecessor (on the path).

Finding the shortest path:

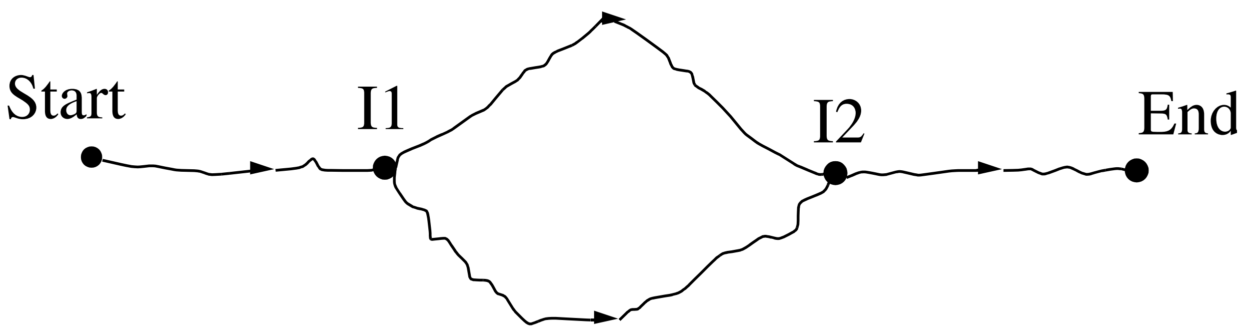
- Whenever the distance of a vertex (supposed to be the shortest one) is modified, the predecessor vertex is stored.

Dynamic Programming

Principle

Any subpath of a shortest path is necessarily a shortest path.

Proof: Otherwise it would be possible to build a shorter path by substituting the shorter subpath.



VI. Analysis of Algorithms (Algorithmique)

Classification of algorithms
Selection criteria
Complexity
Big O notation
Fundamental recurrence relations
Design of algorithms
Incremental algorithms
Greedy algorithms
Divide and conquer algorithms
Dynamic programming
Knapsack problem
Computability and complexity
Undecidable problems
Exponential time problems
Polynomial time problems
NP-complete problems
Satisfiability problem

Algorithms

Sorting →

Searching

- Sequential Searching, Binary Search, Tree Search, Hashing, Radix Searching

String Processing

- String Searching
  - Knuth-Morris-Pratt, Boyer-Moore, Robin-Karp
- Pattern Matching
- Parsing (Top-Down, Bottom-Up, Compilers)
- Compression
  - Huffman Code
- Cryptology

Image Processing

Algorithms

- Geometric Algorithms
  - Intersections
  - Convexity
  - Jordan Sorting
  - Closest-Point Problems
  - Curve Fitting
- Mathematical Algorithms
  - Random Numbers
  - Polynomial Arithmetic
  - Matrix Arithmetic
  - Gaussian Elimination
  - Integration
  - Fast Fourier Transform
  - Linear Programming
- Graph Algorithms $\rightarrow$

Selection Criteria

How to choose an algorithm and/or a data structure representation?

1. Effort for implementing the algorithm:
   1.1. searching the literature
   1.2. programming
   1.3. testing
   1.4. maintenance

2. Resources used for running the algorithm:
   2.1. time (of computation)
   2.2. space (in memory)
   2.3. energy (number of processors)

3. Frequency of use of the algorithm

Complexity

The complexity measure the quantity of resources used by an algorithms as a function of the problem size.

One is especially interested in the trend of the complexity when the problem size becomes large, tends towards infinity.

Worst-case analysis:
complexity for problems the algorithm is in trouble dealing with.

Average-case analysis:
complexity for "average" problems.

Big O Notation

Definition

The big O notation defines equivalence classes of real functions defined on the natural numbers.

$f, g: N^+ \rightarrow R^+$

f belongs to O(g) iff

$\exists\, n_o \in \mathrm{N}, \exists\, c \in R$, such that

$\forall\, n \geq n_o$, $f(n) \leq cg(n)$

Big O Notation

Calculus

1. Transitivity (transitivité)
   If f is O(g) and g is O(h), then f is O(h).

2. Scaling (changement d'échelle)
   If f is O(g), then for all $k > 0$, f is $O(k \cdot g)$.

3. Sum (somme)
   If f_1 is $O(g_1)$ and f_2 is $O(g_2)$,
   then $f_1 + f_2$ is $O(\max (f_1, f_2))$, where
   $\max (f_1, f_2) (x) = \max (f_1 (x), f_2 (x)); \forall x$

4. Product (produit)
   If f_1 is $O(g_1)$ and f_2 is $O(g_2)$,
   then $f_1 \cdot f_2$ is $O(g_1 \cdot g_2)$.

Big O Notation

Example

Show that $2n^3 + 5n^2 + 3$ is $O(n^3)$.

Sum

$$O(2n^3 + 5n^2 + 3) = O(\max(2n^3, 5n^2 + 3)) = O(2n^3)$$

Scaling:

$$O(2n^3) = O(n^3)$$

Transitivity:

$$O(2n^3 + 5n^2 + 3) = O(2n^3)$$

and

$$O(2n^3) = O(n^3)$$

therefore

$$O(2n^3 + 5n^2 + 3) = O(n^3)$$

Big O Notation

Typical Cases

$O(1)$ constant complexity
$O(\log n)$ logarithmic complexity
$O(n)$ linear complexity
$O(n \log n)$ "nlogn" complexity
$O(n^2)$ quadratic complexity
$O(n^3)$ cubic complexity
$O(n^m)$ polynomial complexity
$O(2^n)$ exponential complexity

An algorithm is said to be exponential, or having an exponential performance, if there is no m such that it is of the class $O(n^m)$, i.e. it is not polynomial.

Fundamental Recurrence Relations

1. Loop over the data structure processing each element in turn, then suppress one element from the data structure. Continue until there are no elements left.

$$
\begin{aligned}
C_1 &= 1 \\
C_n &= C_{n-1} + n, \text{ for } n >= 2
\end{aligned}
$$

therefore

$$
\begin{aligned}
C_n &= C_{n-2} + (n-1) + n \\
&\quad ... \\
&= 1 + 2 + ... + n \\
&= (1/2)*n*(n+1)
\end{aligned}
$$

The complexity is therefore of magnitude n^2.

Example: Selection sort.

Fundamental Recurrence Relations

2. Process one element, then divide the data structure in two equal parts without examining the individual elements. Resume on one of the two parts.

$C_1 = 0$

$C_n = C_{n/2} + 1 \quad n \geq 2$

Approximation with $n = 2^m$

$$
\begin{aligned}
C_{2^m} &= C_{2^{m-1}} + 1 \\
&= C_{2^{m-2}} + 2 \\
&\dots \\
&= C_{2^0} + m \\
&= m
\end{aligned}
$$

$n = 2^m$, hence $m = \lg n$, hence

$C_n \approx \lg n$ (or $\log n$)

Example: Binary search.

Fundamental Recurrence Relations

3. Loop over the data structure processing each element in turn, and dividing on the way the data structure in two equal parts. Resume on one of the two parts.

$C_1 = 1$

$C_n = C_{n/2} + n \quad n \geq 2$

Approximation with $n = 2^m$

$$
\begin{aligned}
C_{2^m} &= C_{2^{m-1}} + 2^m \\
&= C_{2^{m-2}} + 2^{m-1} + 2^m \\
&= 1 + 2^1 + 2^2 + \ldots + 2^m \\
&= 2^{m+1} - 1
\end{aligned}
$$

hence

$$C_n = 2n - 1$$

Example: ??

Fundamental Recurrence Relations

4. Loop over the data structure processing each element in turn, and dividing on the way the data structure in two parts. Resume on the two parts (divide-and-conquer).

$C_1 = 1$

$C_n = 2C_{n/2} + n$

traverse n elements

$C_{n/2} + C_{n/2}$: each half

Approximation: $n = 2^m$

$$C_{2^m} = 2 \cdot C_{2^{m-1}} + 2^m$$

$$\frac{C_{2^m}}{2^m} = \frac{C_{2^{m-1}}}{2^{m-1}} + 1 = m + 1$$

$$C_{2^m} = 2^m \cdot (m + 1)$$

hence:

$$C_n \cong n \cdot \log n$$

Example: Quick sort

Algorithm Design (Conception d'algorithmes)

Know the problems impossible to solve on a computer.

Know the problems hard to compute.

Know the classic algorithms.

Search the literature.

Know how to apply design strategies.

Design Strategies

- Incremental algorithms
  (incremental algorithms)

  Insertion sort, linear search.
- Greedy algorithms
  (algorithmes gloutons)

  Selection sort, shortest path by Dijkstra.
- Divide-and-conquer algorithms
  (algorithmes "diviser pour régner")

  Quick sort, binary search, convex hull.
- Dynamic programing
  (programmation dynamique)
- Search with backtracking (recherche avec rebroussement)
- Pruning (élagage)
- "Branch and bound"
- Approximation
- Heuristics (algorithmes heuristiques)

Incremental Algorithms

```
procedure Solve (P: in [out] Problem;
                 R: out Result) is

begin
   R := some evident value;
   while P ≠ empty loop
      Select X in P;
      Delete X in P;
      Modify R based on X;
   end loop;
end Solve;
```

Incremental Algorithms of the First Kind

The selected X is the first one, the most accessible, etc.

The invariant of the loop is of the form:

R is a complete solution of the subproblem defined by the deleted elements.

Example: Insertion sort

- X is the next element to be processed in the remaining sequence.
- The result is the sorted sequence of the elements already processed.

Greedy Algorithms or Incremental Algorithms of the Second Kind

The element X is more carefully selected.

The invariant of the loop is of the form:

R is a part of the complete solution; R will not be changed, but elements will be added to it.

Example: Selection sort.

In order to produce the sequence

(1, 5, 6, 9, 12),

one produces step-by-step the following sequences:

( ), (1), (1, 5), (1, 5, 6), (1, 5, 6, 9), and (1, 5, 6, 9, 12).

Divide-and-Conquer Algorithms

```
procedure Solve (P: in [out] Problem;
                            R: out Result) is
    P1, P2: Problem; R1, R2: Result;
begin
    if Size (P) < = 1 then
        R := straightforward value;
        return;
    end if;
    Divide P into P1 and P2;
    Solve (P1, R1);
    Solve (P2, R2);
    Combine (R1, R2, R);
end Solve;
```

Sometimes the problem is divided into many subproblems.

The algorithm is especially efficient if the division is into two equally-sized halves.

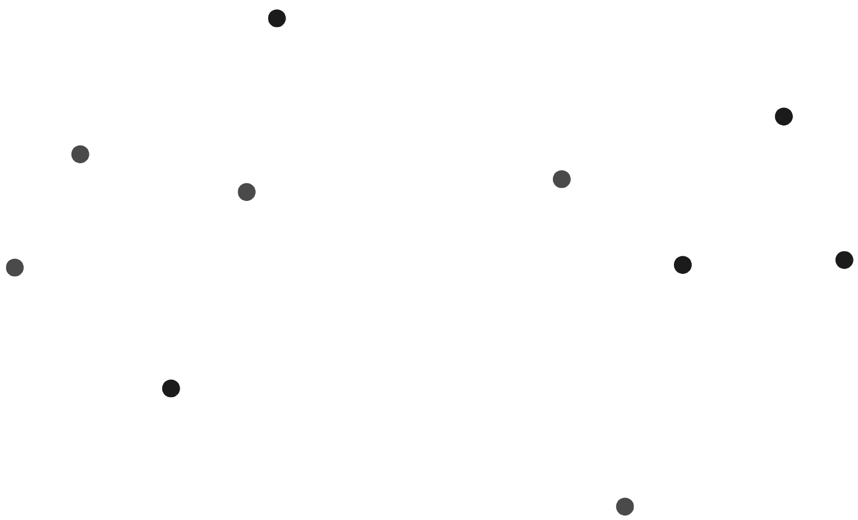
Divide-and-Conquer Algorithms

The difficulty consists in finding the operations Divide and Combine. The easiest way of Dividing will not always allow to Combine the partial solutions into a global solution.

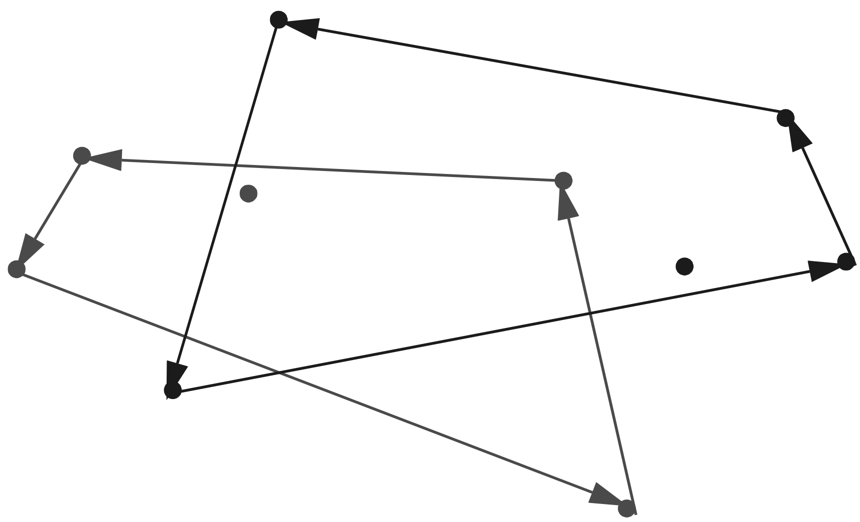
Example: Quick sort

All the effort is put into the Divide operation. The Combine operation is reduced to nothing.

Convex Hull
(Enveloppe convexe)



Divide randomly the points into red and blue ones

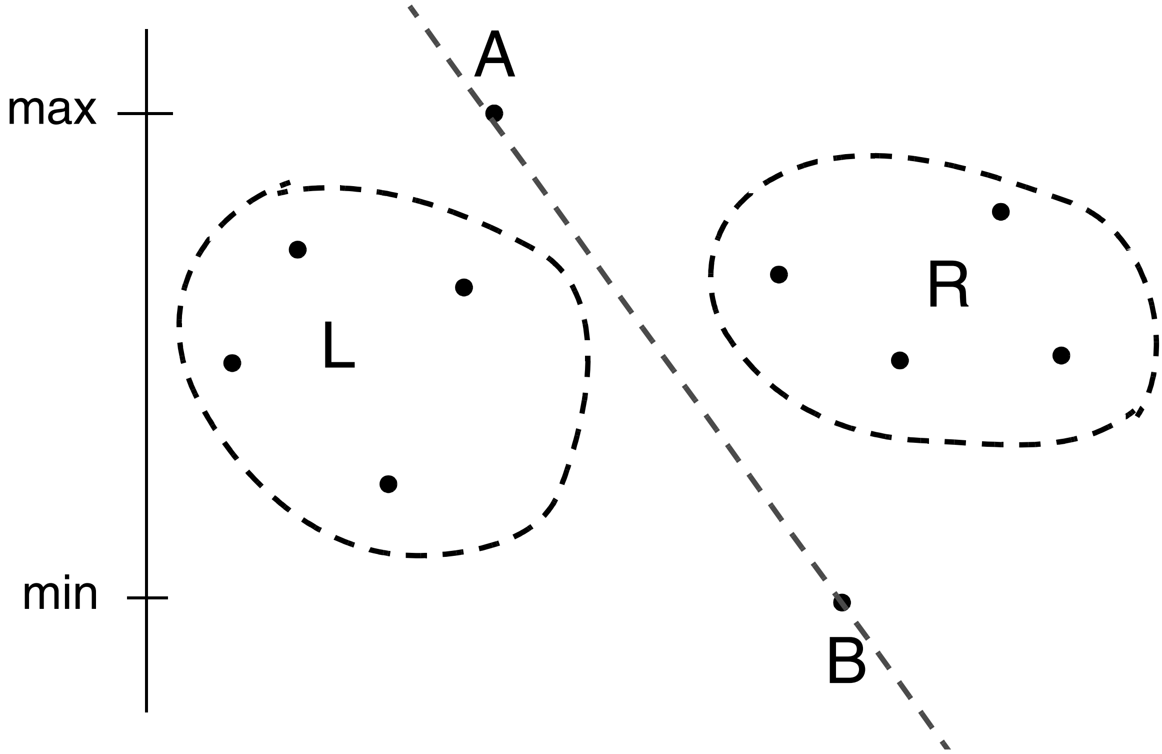


Solve the two subproblems

Red + Blue = ??

Combine: Seems hard!

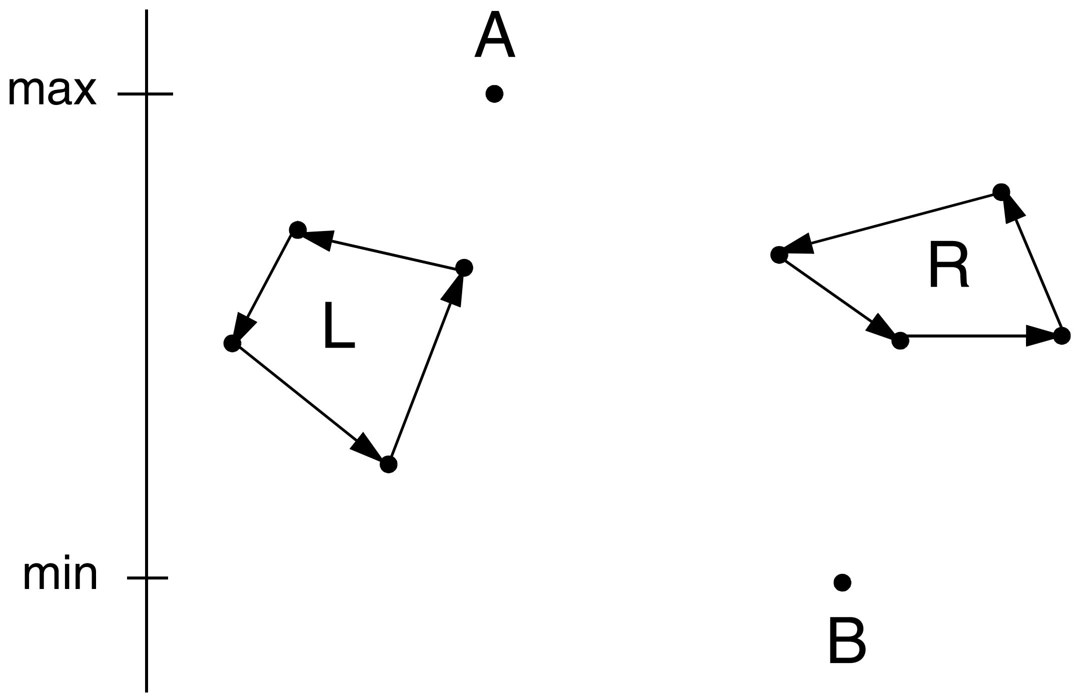
Convex Hull
(Enveloppe convexe)



Divide:

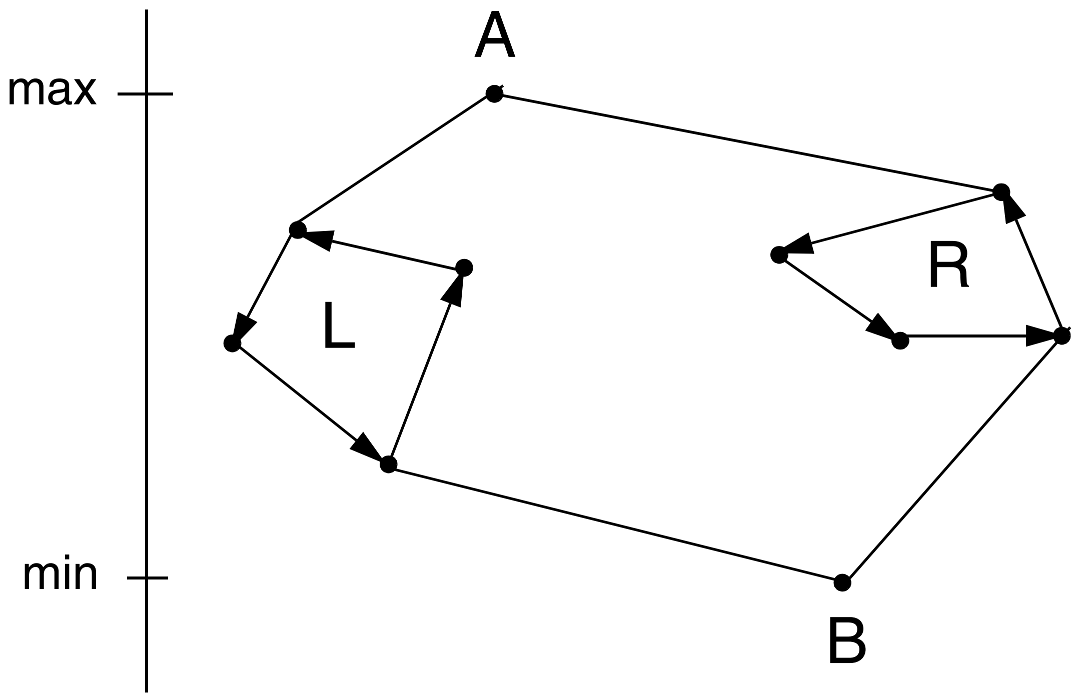
- Find the points with the largest and smallest Y coordinates, called A and B.
- Allocate points to L or R depending on which side of the line joining A and B, left or right, they are.

Convex Hull
(Enveloppe convexe)



Solve L and R

Convex Hull
(Enveloppe convexe)



Combine:

- Connect both A and B to the "right" vertices of the convex hulls of L and R.

Dynamic Programming

Principle of divide-and-conquer:

In order to solve a large problem, it is divided into smaller problems which can be solved independently one from each other.

Dynamic programming

When one does not know exactly which subproblems to solve, one solves them all, and one stores the results for using them later on for solving larger problems.

This principle can be used if:

A decision taken for finding the best solution of a subproblem remains a good solution for solving the complete problem.

Problem of the Knapsack

Capacity of the knapsack: M

List of goods:

Name	A	B	C	D	E
Size	3	4	7	8	9
Value	4	5	10	11	13

Problem

Pack goods of the highest total value in the knapsack, up to its capacity.

Idea of dynamic programming:

Find all optimal solutions for all capacities from 1 to M.

Start with the case where there is only the product A, then the products A and B, etc.

Problem of the Knapsack

k	1	2	3	4	5	6	7	8	9	10	11	12
Obj	0	0	4	4	4	8	8	8	12	12	12	16
Best			A	A	A	A	A	A	A	A	A	A
Obj	0	0	4	5	5	8	9	10	12	13	14	16
Best			A	B	B	A	B	B	A	B	B	A
Obj	0	0	4	5	5	8	10	10	12	14	15	16
Best			A	B	B	A	C	B	A	C	C	A
Obj	0	0	4	5	5	8	10	11	12	14	15	16
Best			A	B	B	A	C	D	A	C	C	A
Obj	0	0	4	5	5	8	10	11	13	14	15	17
Best			A	B	B	A	C	D	E	C	C	E

Problem of the Knapsack

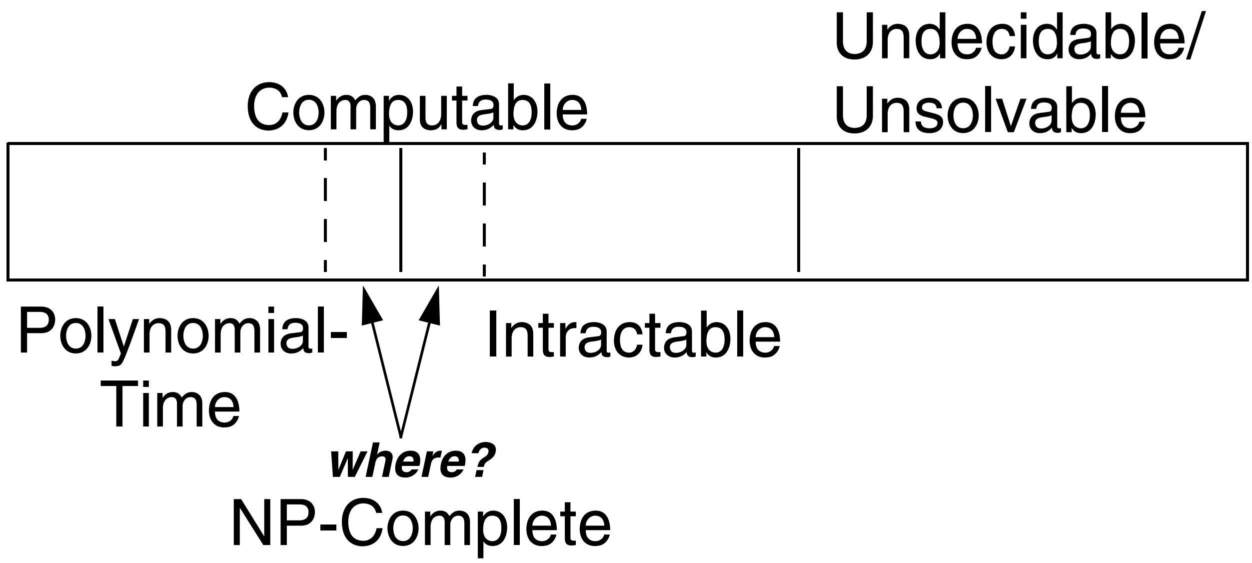
```
type Good is (A, B, C, D, E);
type Table_of_Values is
   array (Good) of Integer;
Size: constant Table_of_Values
      := (3, 4, 7, 8, 9);
Value: constant Table_of_Values
      := (4, 5, 10, 11, 13);
Objective: array (1..M) of Integer
      := (others => 0);
Best: array (1..M) of Good;
```

Problem of the Knapsack

```
for P in Good loop
    for Cap in 1..M loop

        if Cap-Size(P) > = 0 then
            if Objective (Cap)
                < Objective (Cap-Size (P)) + Value (P) then
                    Objectif (Cap) :=
                        Objective (Cap-Taille(P)) + Value (P);
                    Best (Cap) := P;
            end if;
        end if;

    end loop;
end loop;
```

Argument: If P is chosen, the best value is Value (P) plus Value (Cap - Size (P)), which corresponds to the value of the remaining capacity.

Map of Computability and Complexity



Computability:

Whether or not it is possible to solve a problem on a machine.

Machine:

- Turing Machine

Undecidable Problems, Unsolvable Problems

It is impossible to solve the problem by an algorithm.

Examples:

- Halting problem
- Trisect an arbitrary angle with a compass and a straight edge.

Intractable Problems

There is an algorithm to solve the problem. Any algorithm requires at least exponential time.

Polynomial-Time Problems

Size N of a problem:

- Number of bits used to encode the input, using a "reasonable" encoding scheme.

Efficiency of an algorithm:

- Is a function of the problem size.

Deterministic algorithm/machine:

- At any time, whatever the algorithm/machine is doing, there is only one thing that it could do next.

P:

- The set of problems that can be solved by deterministic algorithms in polynomial time.

Non-Deterministic Polynomial-Time Problems

Non-determinism:

- When an algorithm is faced with a choice of several options, it has the power to "guess" the right one.

Non-deterministic algorithm/machine:

- To solve the problem, "guess" the solution, then verify that the solution is correct.

NP:

- The set of problems that can be solved by non-deterministic algorithms in polynomial time.

Non-Deterministic Polynomial-Time Problems

$$P \subset NP$$

To show that a problem is in NP, we need only to find a polynomial-time algorithm to check that a given solution (the guessed solution) is valid.

Non-determinism is such a powerful operation that it seems almost absurd to consider it seriously.

Nevertheless we do not know whether or not:

P = NP ?? (rather no!)

NP-Complete Problems

A problem is said to be NP-complete:

- if it is NP, and
- it is **likely** that the problem is **not** P, and hence
- it is **likely** that the problem is intractable.

Otherwise stated:

- There is no known polynomial-time algorithm.
- It has not been proven that the problem is intractable.
- It is easy to check that a given solution is valid.

It can be shown that:

- ALL NP-COMPLETE PROBLEMS ARE EQUIVALENT.

  (i.e. they may be transformed in polynomial-time each one to another one.)

Satisfiability Problem

Given a logical formula of the form:

$(x_1 + x_3 + x_5) * (x_1 + \neg x_2 + x_4) * (\neg x_3 + x_4 + x_5)$

where the x_i's represent Boolean variables, the satisfiability problem is to determine whether or not there is an assignment of truth values to variables that makes the formula true ("satisfies" it).

- It is easy to check that a given solution satisfies the formula.
- NP-completeness shown by Cook (1971).

NP-Complete Problems

Satisfiability

- Is a Boolean expression satisfiable?

Hamilton circuit

- Does a (un)directed graph have a Hamilton circuit (cycle), i.e. a circuit (cycle) containing every vertex.

Traveling Salesperson problem

Colorability

Is an undirected graph k-colorable? (no two adjacent vertices are assigned the same color)

Graph Isomorphism problem

Rename the vertices so that the graphs are identical.

Longest path (without cycles)

NP-Complete Problems

Knapsack problem

- Fill a knapsack with goodies of best value.
- Given integers i_1, i_2,..., i_n and k, is there a subsequence that sums exactly k?

Integer linear programming

Multiprocessor scheduling

- Given a deadline and a set of tasks of varying length to be performed on two identical processors, can the tasks be arranged so that the deadline is met?

How to Solve Intractable Problems

1. Polynomial-time may be **larger** than exponential-time for any **reasonable** problem size:

- $n^{\lg\lg\lg n}$ is less than n^2 for $n < 2^{16} = 65536$
- $n^{\lg\lg\lg n}$ is less than n^3 for $n < 2^{256} \approx 10^{77}$

2. Rely on "average-time" performance. The algorithm finds the solution in some cases, but does not necessarily work in all cases.

3. "Approximation"

The problem is changed. The algorithm does not find the best solution, but a solution guaranteed to be close to the best (e.g. value $\geq$ 95% of best value)