IFT2015 :: A17 Miklós Cs˝urös 22 September 2017

4. Recursive Structures: Lists and Trees

T HE LISTS AND TREES are fundamental structures defined by recursion. The base case
(typically by a null reference). The recursive case captures the self-resemblance of the structure with
minus a member: a non-empty list is formed by a head and sub-list of its successors, and a non-empty tree is
formed by its root and a set of subtrees.

The natural implementation of operations in such a structure exploits recursion. The recursive definition
also translated into a specific representation: the structure is organized by **nodes** each of which contains data
associated with a single element, and also links to other nodes (references to neighbors on the list or relationships
in the tree). In **endogenous** implementation, data is handled in concert with links: content is
inseparable from the container. In object-oriented languages, this style of programming corresponds to the manipulation of
objects of nodes with the references stored to them as instance variables. But it is also possible to manipulate
donnnées units séparamment units of the structure and design an **exogenous** implementation (eg, based on
the exchange of contents between the boxes of a table).

4.1 Recursive Structures

The fundamental principles of data organization, in particular the sequential or hierarchical arrangement,
structures that can be defined recursively. The lists (Def. 4 .1) and trees (Def. 4.2) are types
aggregated "self-similar". A non-empty list is the combination of its head and the sub-list after. A non-empty tree is
a collection of subtrees attached to a common root.

Definition 4.1. *The* ***list*** *is a recursive structure built by the application of rules.*

list → *<external node>* = null *empty list*

→ list *(<internal node>,* list) *(Head, successors)*

empty list ***non-empty list***

head *this is a list*

F IG. 1: A list is either a reference null (a single external node), or a pair formed by an internal node (the head) and from another list.

Definition 4.2. *A* ***rooted tree*** *(or "tree structure") is a recursive structure constructed according to the following rules.*

tree → *<external node>* *empty tree* = null

→ tree *(<internal node>* $\underbrace{\text{tree} \ldots, \text{tree}}_{d>0 \text{ times}}$) *(root,* for *children)*

empty tree

root

it is a treeit is a tree

it is a tree

non-empty tree

F IG. 2: A rooted tree is either null (external node), or it is composed of one internal node (the root) and a set of rooted trees (the children).

4.2 Linked list

The structure called **linked list** [1] *(linked list)* is a list of items kept in a node that also contains one or two links on the next node [2] and / or prior (Fig. 3). The list is identified by its first node, or the **head** *(head)*. The last internal node is called the **tail** *(tail)* of the listing.

Nodes on a standard linked list have two variables: one for store the data associated with the node and another that references the head of the sublist after (x.data and x.next). The nodes on a **list doubly** have **linked** three variables: one for storing data associated with the node and two others to store references to neighboring nodes. On a **circular list,** tail and head are related (in place of null references linear lists).

[3] With generic Java can enforcer data typing placed at compile nodes. In the implementation below, the class of the node is parametrized by the type of data that can be stored there (type Data).

```
/ **
 * List node with typed payload
 * @param <Data> type of payload
 * /
class ListNode <Data>
{
    private final Data data;   // inseparable
    private ListNode <Data> next;      // next item
    ListNode (Data data) // new node without successor
    {

        this.data = data; this.next = null;
    }
    Data getData () {return this.data;}
    ...
}
```

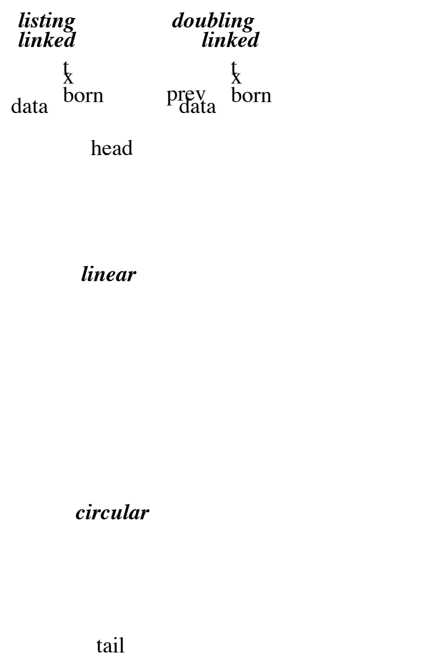
Parametric classes. Parameterized type [4] is declared with C <A, B, ...> where C is the generic class with parameters A, B which are types (classes) themselves. Important aspects of parametric types in Java.

- The parameters in the class declaration belong to an instance and do not exist in a static context.
- The arguments prametrisés types are checked during compilation but the information is not available at runtime. So,
  - You can not instantiate an object with the standard setting (**new** A ()) do not work),
  - We can not check whether an object is a member (x **instanceof** A does not work), and
  - **GetClass** () gives no information on the arguments used during the instantiation.

  For the same reason, arrays of parametrized types

1 (en) : linked list

2 Note 4. Set 1 establishes the concepts of "Next" and "previous" with precision, recurrence (the head is before the nodes on the sublist in the recursive case)



F IG. 3: Linked lists with common nodes bearing one or two links. If the list is circularized, a reference to the *tail:* the head is the next node.

3 Java tutorial (*Java generics*)

4 Java Language Specification: *Parametrized Types*

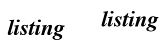
Page 3

(like LinkedList <String> [10]) because typing can not be to the members of the table.

- The legacy does not transfer through generic classes: although E F is a sub-class LinkedList <E> is not a subclass LinkedList <E>.

Linked List Operations

The linked list supports the route and the local manipulation of the order. It is simple to insert or remove the head or after a given node. The cularisation gives access to the tail (and head). A doubly linked list is used if navigation is required in both directions on the list, deletion and insertion before a node because we must change the references in the previous node (Fig. 4) .



F IG. 4: Operations with constant time

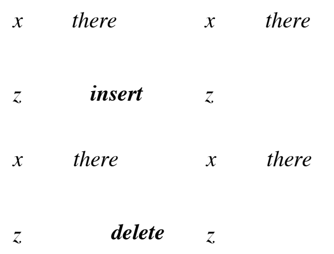
operations	*linked linen*	*Circ.*	*doubling linked linen*	*Circ.*
access to the head TA stack		+		
access to the tail TA tail	-	+	-	+
concatenate next insert / delete after		+		
previous insert / delete before	-		+	
overthrow		-		+

the double-linked list. A circu- the reversal implementation of list, with the trick of keeping a bit next to it to interpret .next / .prev as pro- chain / previous or vice versa.

Insert and delete. Add a new head or "cut" the head of a list is trivial. Inserting or deleting after the head requires the assignment of two references or a reference (Fig. 5) :

```
class ListNode <Data>
{
    private Data data;
    private ListNode <Data> next;
    ...
    void insertNext (ListNode <Data> y) {
        LisrNode <Data> z = next;
        y.next = z;
        next = y;
    }

    void deleteNext () {
        ListNode <Data> y = next;
        ListNode <Data> z = y.next;
        next = z;
    }
    ...
```



F IG. 5: Insertion and removal after the head.

Recursive implementations. One can exploit the recursion of the structure in the implementation by decomposing an operation on a non-empty list:

- work on the head node
- work on sub-list after the head [by recursive call]
- combination of results on the head and on the successor list [better from delegator to recursive call for terminal recursion]

```
class ListNode <Data>
{
    private Data data;
    private ListNode <Data> next;
    ...
    / * access to nodes by index * /
    ListNode <Data> getNode (int i)
    {
        if (i = 0) return this;
        else return next.getNode (i-1);
    }
    /* research */
    boolean contains (Data key)
    {
         return data.equals (key)
            || (next! = null && next.contains (key));
    }
    / * ... * /
```

```
/ * ... * /
/ * suppression by index; returns the new head * /
ListNode <Data> deleteAt (int i)
{
    if (i == 0) {return node.next;}
    else {next = next.deleteAt (i-1); return this;}
}
/ ** length of the list
  * @param x head of the list (possibly null)
  * @param L 0 at first call
* /
static int length (ListNode <?> x, int L)
{
    if (x == null) return L;
    return length (x.next, L + 1);
}
...
} // class ListNode
```

Implementations iterative. The list accommodates iterative implementations great too.

```
class ListNode <Data>
{
    private Data data;
    private ListNode <Data> next;
    ...
    static <D> ListNode <D> getNode (ListNode <D> head, int i)
    {
        ListNode <D> node = head;
        while (i> 0) {node = node.next; --i;}
        return node;
```

```
static boolean contains (ListNode <?> head, Object key)
{
    ListNode <?> Node = head;
    while (node! = null &&! node.data.equals (key))
        node = node.next;
    return (node! = null);
}
static int length (ListNode <?> x)
{
    int L = 0; for (; x! = null; x = x.next) L ++;
```

}   } return L;

sentinels

A guard [5] may be used to denote the head and / or tail.

5 (in) : Sentinel

Definition 4.3. *A **sentinel** is a dummy member in a data structure.*

empty list

Advantage: code lighter, execution a little faster. Disadvantage: one
→ No more node lists the total length require n + nodes instead
s.

sentinel

F IG. 6: Linear and circular lists with sentinel.

Page 5

4.3 Trees

According to Definition 4.2, a rooted tree T is a structure defined on a set of nodes that

1. is an **external node,** or
2. consists of an **internal node** called the **root** r, and a set of rooted trees **(children)**

F IG. 7: Tree rooted.

w relative

child

root

subtree of
w root

Definition 4.4. *An **ordered tree*** T *is a structure defined on a set of nodes that*

*1. is an **external node,** or*

*2. consists of an **internal node** called the **root** r, and the trees* $T_0, T_1, T_2, \ldots, T_{d-1}$.
The T_i *root is call'd the **child** labeled by r* i *or* r i *th child.*

*The **degree** (or **arity)** of a node is the number of children: external nodes are of degree 0. The degree of the tree is the maximum degree of its nodes. A tree is k-ary an ordered tree in which each internal node has exactly* k *children. In particular,* [6] *a **binary tree** is an ordered tree where each internal node has exactly 2 children: the left and right subtrees.*

6 (en) : Binary arbr

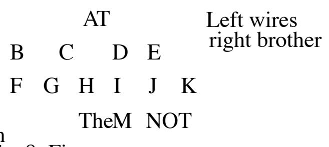
Representation of the nodes of the tree

Tree = set of objects representing nodes + parent-child. In general, parents and children are any node. Typically, external nodes do not carry data, and are simply represented by null links. If the tree is of arity k, they can be stored in an array of size k.

```
class TreeNode // internal node in binary tree
{
    TreeNode parent; // null at the root
    TreeNode left;   // left child, null = external child
    TreeNode right;// child right, null = external child
    // ... other information
}
class MultiNode // internal node of arbitrary degree
{
    MultiNode parent; // null at the root
    MultiNode [] children; // children; children.length = arity
    // ... etc
}
```

Page 6

First-child, next-sibling. If the arity of the tree is not known in advance, can use a list to store children: by nodding the head of the list of his children, as well as the reference to the next node on his own list of siblings, **first son** named representation is obtained, **next brother** *(first-child, next-sibling),* c. Fig. 8 . (The first son is the head of the list of children, and the next brother is the pointer to the next node on the list of children.) In this representation it is necessary to store the external nodes explicitly (because the external node can have an internal node for right).

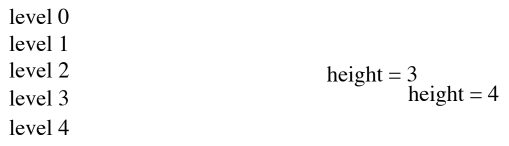


F IG. 8: First-son / next brother.

Theorem 4.1. *There is a correspondence one to one between binary trees with* n *internal nodes and ordered trees with* n *nodes (internal and external).*

Demonstration. One can interpret "first son" as "left child", and "Next brother" as "right child" to get a single binary tree which corresponds to an arbitrary ordered tree, and vice versa. ■

Properties of nodes



F IG. 9: Height and level.

Level (level / depth) of a node u: length of the path that leads to u from of the root

Height (height) of a node u: u maximum length of a path to an external node in the subtree of u

Tree height: height of the root (= maximum level nodes)

Path Length (internal / external) (internal / external path length) of the sum levels of all nodes (internal / external)

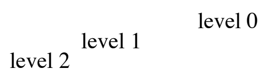
$$\text{height}[x] = \begin{cases} 0 & \text{if x is external;} \\ 1 + \max_{y \in x.\text{children}} \text{height}[y] & \end{cases}$$

$$\text{level}[x] = \begin{cases} 0 & \text{if x is the root (x.parent = null);} \\ 1 + \text{level}[x.\text{parent}] & \text{otherwise} \end{cases}$$

Theorem 4.2. *A binary tree with* n *external nodes contain* $(n - 1)$ *nodes internal*.

A **complete binary tree** of height h: ilya $_i$ 2 nodes at each level $i = 0, ..., h - 1$. "fills" the levels from left to right.

Page 7



F IG. 10: A complete binary tree.

level 3

Theorem 4.3. *The height* h *of a binary tree with* n *internal nodes is bounded by*
⌈lg (n + 1) ⌉ ≤ h ≤ *n*.

Demonstration. A shaft height h = 0 contains only a single node f
dull (n = 0), and the terminals are correct. For h> 0 is defined as $_k$ n
the number of internal nodes at level k = 0, 1, 2,. . . H - 1 (there is no
internal node at level h). One year = C $_{h-1}$ $_{k=0}$ n $_k$. As n $_k$ ≥ 1 for all
k = 0, ..., h - 1, we have that n ≥ C $_{h-1}$ $_{k=0}$ 1 = h. For an upper bound,
is used as d $_0$ = 1, and n $_k$ ≤ 2n $_{k-1}$ for all k> 0. Accordingly,
n ≤ C $_{h-1}$ $_{k=0}$ 2 ^{k} = ^{h} 2 - 1, where h ≥ lg (n + 1). The evidence also shows
end shafts: a knotted string for h = n, and a binary tree com-
plete. ■

course

A route visits all the nodes of the tree. In a **path prefix**
(Preorder traversal), each node is visited before its children are visited.
We thus compute properties with recurrence towards the parent (as level).
In **postfix Course** *(postorder traversal),* each node is visited after
that his children are visited. Recursive properties are thus computed
to children (as height).

In the following algorithms, an external node is null, and each node
internal N has the variables N.children (if ordered tree), or N.left and
N.right (if binary tree). The tree is stored by a reference to its root root.

```
Algo P ARCOURS - PREFIX (x)
1 if x = null then
2     «Visit» x
3     for y ∈ x.children do
4         P ARCOURS - PREFIX (y)
```

```
Algo P ARCOURS - postfixed (x)
1 if x = null then
2     for y ∈ x.children do
3         P ARCOURS - postfix (y)
4     «Visit» x
```

```
Algo DUCATIONAL N (x, n)        // fill level [···]
// x is parent to level n
// call initiel x = root and n = - 1
N1 if x = null then
N2     level [x] ← n + 1               // (prefix visit)
N3     for y ∈ x.children do
N4         N EVEL (y, n + 1)
```

```
Algo AUTHOR H (x)               // returns height x
← H1 max - 1                    // (maximum height of children)
H2 if x = null then
H3     for y ∈ x.children do
H4         h ← AUTHOR H (y);
H5         if h> max max Then ← pm
H6 return 1 + max               // (postfix visit)
```

Page 8

In an **inorder traversal** *(inorder traversal),* we
visit each knot after her left child but
before her right child. (This course is only
on a binary tree.)

```
Algo P ARCOURS - INFIX (x)
1 if x = null then
2     P ARCOURS - INFIX (x.left)
3     «Visit» x
4     P ARCOURS - INFIX (x.right)
```

A prefix route can also be made using a battery. If instead of the stack, we use a tail, then we get
a **course by level.**

```
Algo P ARCOURS - BATTERY
1 initialize the P battery
2 P.push (root)
3 while P = 3
4     x ← P.pop ()
5     if x = null then
6         «Visit» x
7         for y ∈ x.children do P.push (y)
```

```
Algo P ARCOURS - LEVEL
1 initialize queue Q
2 Q.enqueue (root)
3 while Q = 3
4     x ← Q.dequeue ()
5     if x = null then
6         «Visit» x
7         for y ∈ x.children do Q.enqueue (y)
```

Syntax tree

*

notation infix: 2 * (3 + 7)
notation prefix: * 2 + 3 7

An arithmetic expression may be represented by a **tree syn-**
taxic. Different paths of the same tree lead to representations
different from the same expression.

An arithmetic operation op b is written in **Polish notation**
7 or **inverse** notation "postfix" by ab op. Advantage: no brackets!
Examples: 1 + 2 + 12 → (37) → 8 * 37-8 *. Such an expression
rates using a stack: op ← pop () b ← pop (), a ← pop (), c ←
op (a, b), push (c). The code is repeated while there is an operator at the top. AT
the end, the stack contains only the numerical result. The evaluation
a postfixed path of the syntactic tree.

2 + notation postfix: 2 3 7 + *

3 7

F IG. 11: Syntax tree. The tree shows the application of rules in a grammar for formal expressions: E → E + E | E * E | *number.*

7 (en) : Ratings infix, prefix, Polish and postfixed

```
VAL algorithm E (x)                // (évalation of the parse tree with root x)
E1 if x has no children then return its value // (a constant)
E2 else // (x is an operation op of arity k)
E3      for i ← 0, ..., k - 1 do f i ← E VAL (x.children [i])
E4      return the result of the operation op with the operands (f 0, f 1, ..., f k - 1 )
```

The prefix notation is typically used for calling procedures, functions, or methods in languages
popular programming such as Java and C. At the same time, arithmetic and logical operations are typically
written in infix notation. The **PostScript** language uses postfix notation everywhere, even in instructions
control. As a result, the interpreter connects on stacks in execution. The b sub code {5 2 20 30 40 lineto moveto} if
draws a line between the points (3, 20) and (30, 40) if b is true. In Javaesque we could write if (b) {moveTo (5-2.20); lineTo (30, 40)}.
Lisp uses prefix notation mandatory parentheses. Consequently, the lists are basic
mental ("Lisp" = *List Processing Language)* during execution: a list is formed of a head (because for Lispéens) and the list
successors (cdr for Lispéen).

(with-canvas (: width 100: height 200) ((moveto (- 5 2) 20) (lineto 30 40)