Algorithmics

Éric Renault — Qin Wei

Organization

- Tutorial 1 — 14/12, 14h-17h30 — INT A07.01
  - Recursion
  - Stacks
- Tutorial 2 — 04/01, 14h-17h30 — INT A07.01
  - Queues
- Tutorial 3 — 11/01, 14h-17h30 — INT A07.01
  - Lists
  - Sorted lists
- Tutorial 4 — 18/01, 14h-17h30 — INT A07.01
  - Binary trees
- Tutorial 5 — 25/01, 14h-17h30 — INT A07.01
  - Binary Search Trees
  - Trees
- Tutorial 6 — 01/02, 14h-17h30 — INT A07.01
  - Sorts

Algorithmics

Éric Renault

Institut National des Télécommunications
9, rue Charles Fourier
91011 Évry Cedex, France

Contents

Introduction

- This course does not aim at doing C again
- It aims at presenting :
  - concepts to perform efficiently classical operations
  - important data structures
- The most important here is not syntax but semantic

Pointers

- Usually, when one wants to specify a variable, one provides the value of the variable
- This implies two main drawbacks :
  - it takes lots of time to copy large-size variables
  - as a copy is provided, modification on the copy are not forwarded to the original
- A solution is to use pointers :
  - no extra copy and the size of a pointer is very small
  - modifications are performed on the original information

Implementation

Three different programming styles :

- Functional

```
type function ( type data, parameters... )
```

- Procedural

```
void function ( type * data, parameters... )
```

- Object

```
void type :: function ( parameters... )
```

- Concepts are presented using the functional style

Recursion

Definition

Definition :

- The recursion is the ability for a function to call itself

Why ?

- Some problems are easier to understand this way

Problem :

- One must be very careful as it is very easy to create programs which never stop

Examples

Some famous mathematic series are defined this way :

- The Fibonacci series :

$$\begin{aligned} F_0 &= F_1 = 1 \\ F_{n+2} &= F_{n+1} + F_n \qquad \forall n \in \mathbb{N} \end{aligned}$$

- The series used for the Syracuse conjecture :

$$\begin{aligned} u_0 &= k \qquad k \in \mathbb{N} \\ u_{n+1} &= \begin{cases} u_n/2 & \text{if } u_n \text{ is an even number} \\ 3 \times u_n + 1 & \text{if } u_n \text{ is an odd number} \end{cases} \end{aligned}$$

Never-ending programs avoidance ?

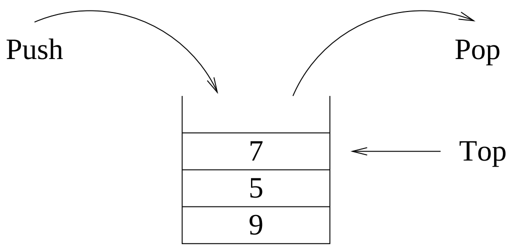
- When writing a recursive function, first introduce the stop conditions and then operations to perform
- A typical recursive function should look like :

```
type my_function ( types my_args ) {
    if ( a_stop_condition )
        return ... ;

    do_the_job ... ;

    return ... ;
}
```


STACKS

Definition

- Set of elements of the same type
- The number of elements varies and is finite (≥ 0)
- LIFO (Last In First Out) management

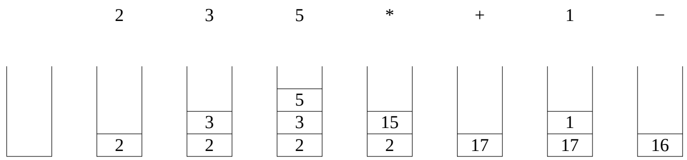


Operations

- `stack new ( ) ;`
- `boolean is_empty ( stack ) ;`
- `stack push ( stack, type ) ;`
- `stack pop ( stack ) ;`
- `type top ( stack ) ;`

Applications

- Evaluation of mathematics expressions : example of a postfix expression (reverse-polish notation)



- Execution of recursive functions : parameters and local variables are stored on the application stack

Array representation

```
# define MAX 10

typedef struct {
  type value [ MAX ] ;
  int top ;
} stack ;

stack s ;
```

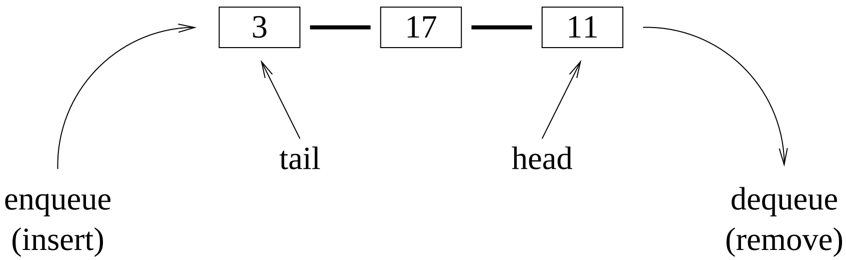

Pointer representation

```
struct element {
  type value ;
  struct element * next ;
} ;

typedef struct element * stack ;

stack s ;
```


QUEUES

Definition

- Set of elements of the same type
- The number of elements varies and is finite (≥ 0)
- FIFO (First In First Out) management



Operations

- `queue new ( ) ;`
- `boolean is_empty ( queue ) ;`
- `queue enqueue ( queue, type ) ;`
- `queue dequeue ( queue ) ;`
- `type head ( queue ) ;`

Applications

System programming :

- Shared printers management
- Processor allocation to executing programs
  $\rightarrow$ Queue with priority

Complex data-structure path :

- Width first course

Array representation

```
# define MAX 10

typedef struct {
  type value [ MAX ] ;
  int head, tail ;
} queue ;

queue q ;
```

Pointer representation

```
struct element {
  type value ;
  struct element * next ;
} ;

typedef struct {
  struct element * head, * tail ;
} queue ;

queue q ;
```


LISTS

Definition

Definition :

- Set of elements of the same type
- The number of elements varies and is finite (≥ 0)
- No order

Caracteristics :

- Access to any element
- Positioning according to criteria
- Access / insert / remove an element

Operations

- `list new ( ) ;`
- `boolean is_empty ( list ) ;`
- `int size ( list ) ;`
- `list enlist ( list, type ) ;`
- `list delist ( list ) ;`
- `type value ( list ) ;`
- `list next ( list ) ;`

Applications

Research in a set of values :

- $O(n)$ or $O(\log_2 n)$, depends upon the implementat.

Polynomial representation :

- a polynomial is a list of monomial
- a monomial is a couple (coef, degree)

Text representation :

- a text is a list of lines
- a line is a list of words

Representation of the structure of a graph

Mono-directional representation

```
struct element {
  type value ;
  struct element * next ;
} ;

typedef struct element * list ;

list l ;
```


Bi-directional representation

```
struct element {
  type value ;
  struct element * next, * previous ;
} ;

typedef struct {
  struct element * first, * last ;
} list ;

list l ;
```


SORTED LISTS

Definition

- A sorted list is a list where the order of elements is the order of the values stored in the list
- It is considered a value is never stored twice
  $\rightarrow$ This is always possible

Operations

These operations are specific to sorted lists :

- `list enlist ( list, type ) ;`
- `list delist ( list, type ) ;`
- `boolean is_element ( list, type ) ;`
- `? elements_perform ( list, function ) ;`

Research in a sorted list

Input :

- a sorted list l and a value v

Output :

- a boolean : true if v is in l and false in the other case

Algorithm :

- Try each element v' in l until one of these conditions is true :
  1. the end of the list $\rightarrow$ not found
  2. $v' > v \rightarrow$ not found
  3. $v' = v \rightarrow$ found

Binary Trees

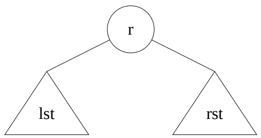
Definition

Definition :

- Set of elements (called “nodes”) of the same type
- The number of elements varies and is finite (≥ 0)

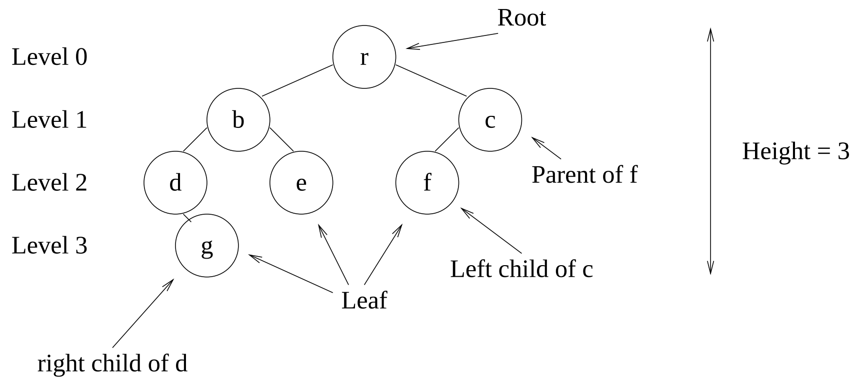
A binary tree is :

- empty : $a = ()$
- composed of one root r and two sub-trees lsb and rsb : $a = (r, lst, rsb)$



Structure

$$a = (r, (b, (d, (), (g, (), ())), (e, (), ())), (c, (f, (), ()), ()))$$



Some definitions

Arity : number of non-empty sub-trees

- the arity of a leaf is 0

Path : sequence of nodes $(n_0, n_1, ..., n_p)$ where n_{i-1} is the parent of n_i

- p is the length of the path

Level (or height) of a node : the length from the root to that node

- the level of the root is 0

Height of a tree : the maximum height for all nodes

- as a definition, the height of an empty tree is -1

Operations

- `btree new ( ) ;`
- `boolean is_empty ( btree ) ;`
- `type root ( btree ) ;`
- `btree build ( type, btree, btree ) ;`
- `btree left ( btree ) ;`
- `btree right ( btree ) ;`
- `boolean is_element ( btree, type ) ;`

Applications

Representation of arithmetics expressions :

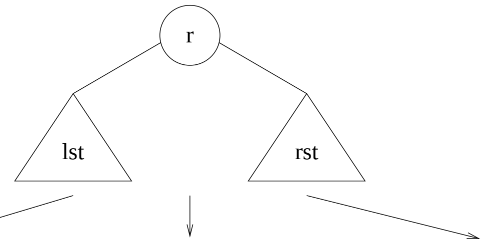
- Various way to run over the binary tree

Compression methods :

- Quad-tree
- Huffman code

Depth first course

a = (r, lst, rst)



Prefixed course
. Perform the root
. Prefixed course of lst
. Prefixed course of rst

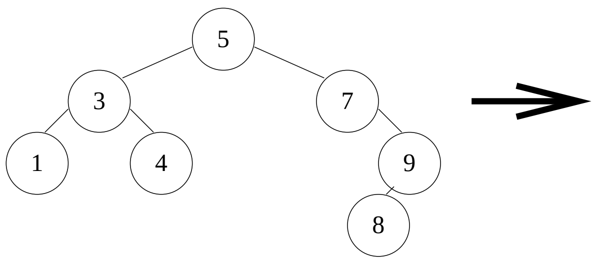
Unfixed course
. Unfixed course of lst
. Perform the root
. Unfixed course of rst

Postfixed course
. Postfixed course of lst
. Postfixed course of rst
. Perform the root

Width first course

Principle :

- The tree is ran over level after level
- Each level is ran over from left to right



(5, 3, 7, 1, 4, 9, 8)

Array representation

```
# define MAX 10

typedef struct {
  type value ;
  int free ;
} btree [ MAX ] ;

btree b ;
```

- lst and rst of node i are $2i$ and $2i + 1$

Pointer representation

```
struct element {
  type value ;
  struct element * left, * right ;
} ;

typedef struct element * btree ;

btree b ;
```


BINARY SEARCH TREES

Definition

Definition :

- A binary search tree (BST) is a binary tree in which nodes are organized according to the relation between the values of the tree

A binary search tree is :

- empty : $a = ()$
- not empty : $a = (r, lst, rst)$

  Let v be the value of the root
  - If lst is not empty, then lst is a BST and any value in lst is less than v
  - If rst is not empty, then rst is a BST ans any value in rst is greater than v

Operations

```
bst add ( bst, type ) ;
bst remove ( bst, type ) ;
?  MODE_run_over ( bst, function ) ;
```

where `MODE` is :

- `prefixed`
- `unfixed`
- `postfixed`

Plus operations on binary trees

Applications

Research in a set of values :

- $O(\log_2 n)$ in a binary search tree

Sort of a set of values :

- Unfixed course of a binary search tree

Research

```
boolean is_element ( bst b, type v ) {
  if ( is_empty ( b ) )
    return false ;
  if ( root ( b ) == v )
    return true ;
  if ( root ( b ) > v )
    return is_element ( left, v ) ;
  return is_element ( right, v ) ;
}
```

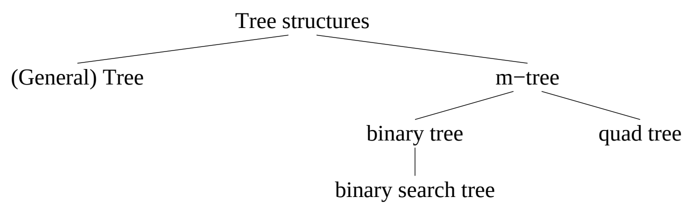

TREES

Definition

Definition :

- Same as a binary tree except that the number of children is not limited to two
- An m-tree is a tree where nodes :
  - are empty
  - have exactly m children

Classification :



Representation

- When two nodes have the same parent, they are "brothers"

- Any general tree may be stored using a binary tree. To do so, for each node :
  - the lst of the node in the binary tree is the first child of the node in the general tree
  - the rst of the node in the binary tree is the next brother of the node in the general tree

Applications

Filesystem :

- the root, the nodes are the directories and the leaves are the regular files

Syntaxical analysis of a text :

- syntaxic tree

Representation digitalized BW images :

- quad-tree (4-tree)

Database with index :

- balanced trees (btrees)

SORTS

Definition

- Sorting is the action to order elements according the an order relation (typically like $\geq$ for numbers)

- Two kinds of methods :
  - internal methods : bubble sort, sort by insertion, quick sort...
  - external methods : merge sort...

Bubble sort

- Suppose elements are stored in a vertical array
- Elements with the smallest keys are lighter and, like bubbles, are moving to the surface
- One performs successive passage on the array from bottom to top
- At each step, if two elements are not in the correct order, they are swapped
- At the end of the first passage, the element at the top is the lightest one ; at the end of the second passage, the second element at the top is the second lightest one...
- Complexity : $O(n^2/2)$

Sort by insertion

- During step i, the i-th element is inserted at the good position between elements 1 and $i-1$
- After step i, each element between 1 and i are sorted. This is called the "invariant"
- Complexity : $O(n^2/2)$

Quick sort

- At each step, choose a pivot
- Then reorganize the list so as to have elements smaller than the pivot on the left and larger ones on the right
- Note that the best pivot is the one that separates the list of elements in two sub-lists of the same size
- Then take each sub-list and restart the operation
- The sort is locally completed when a list is composed of 1 or 2 elements
- Complexity : $O(n \log n)$

Merge sort

- This is a external method
- Suppose two arrays have been sorted using an internal method
- It is not necessary to re-sort both arrays
- The result can be obtained by removing the element which is the smallest regarding both arrays and repeating the operation until both arrays are empty
- Complexity : $O(n+m)$