Binary Trees

Binary search trees - Applications - Implementation

D.E ZEGOUR

Ecole Supérieure d'Informatique

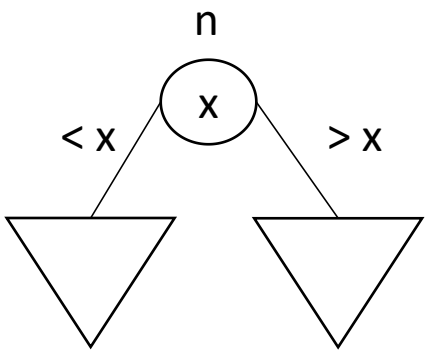
ESI

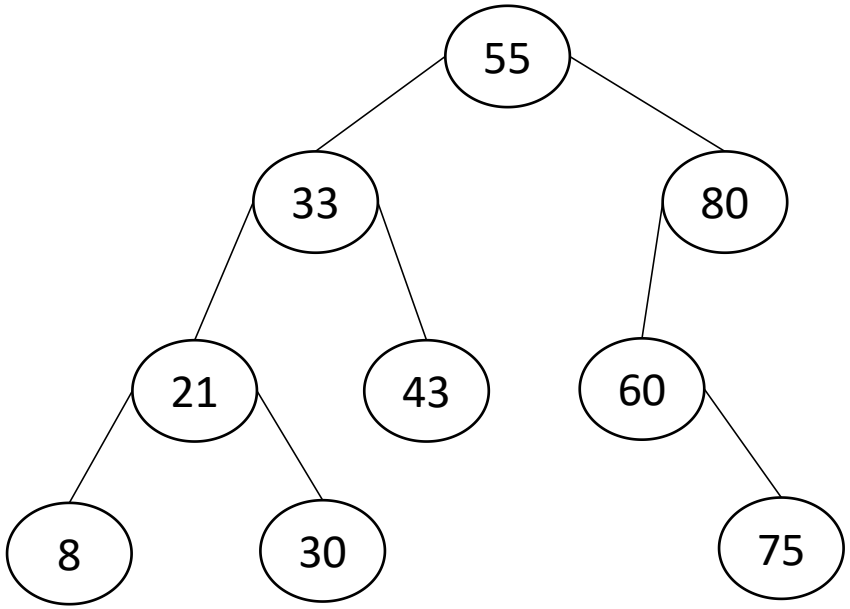
Binary trees

Binary search trees

represent sets whose elements are arranged according to an ordering relation denoted by '<' (in the strict sense)

- all elements stored in the left subtree of a node n containing x are strictly less than x
- all elements in the right subtree of a node containing x are strictly greater than x.





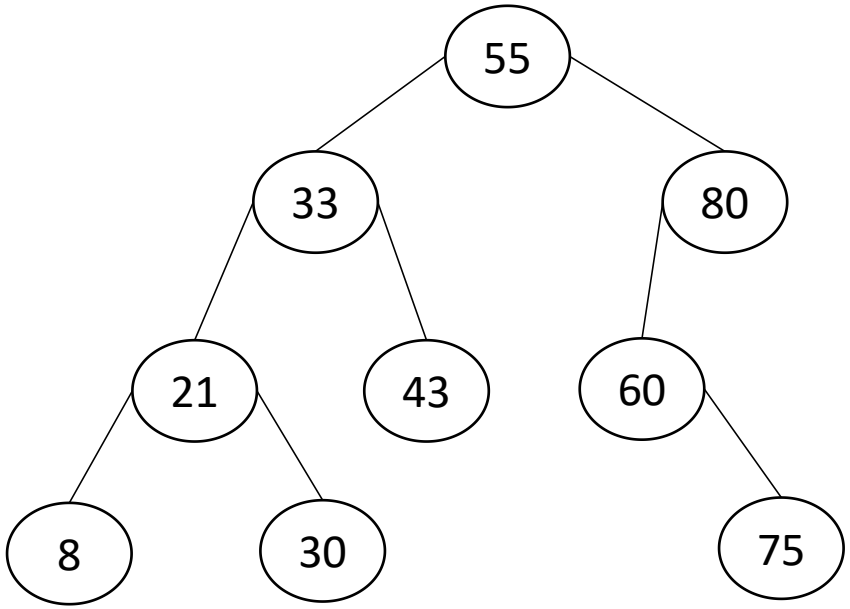
Property : the inorder traversal of a binary search tree produces an ordered list of all its elements.

Inorder traversal: 8, 21, 30, 33, 43, 55, 60, 75, 80

Binary trees

Binary search trees : Searching

```
{Searching for X in BST with root A}
Search(A, X)
  IF A = NIL
        Search:= FALSE
  ELSE
       IF X = Node_value(A)
             Search:= TRUE
          ELSE
             IF X < Node_value(A)
                   Search:= Search( LC(A), X)
                ELSE
                   Search:= Search( RC(A), X)
             ENDIF
        ENDIF
ENDIF
```



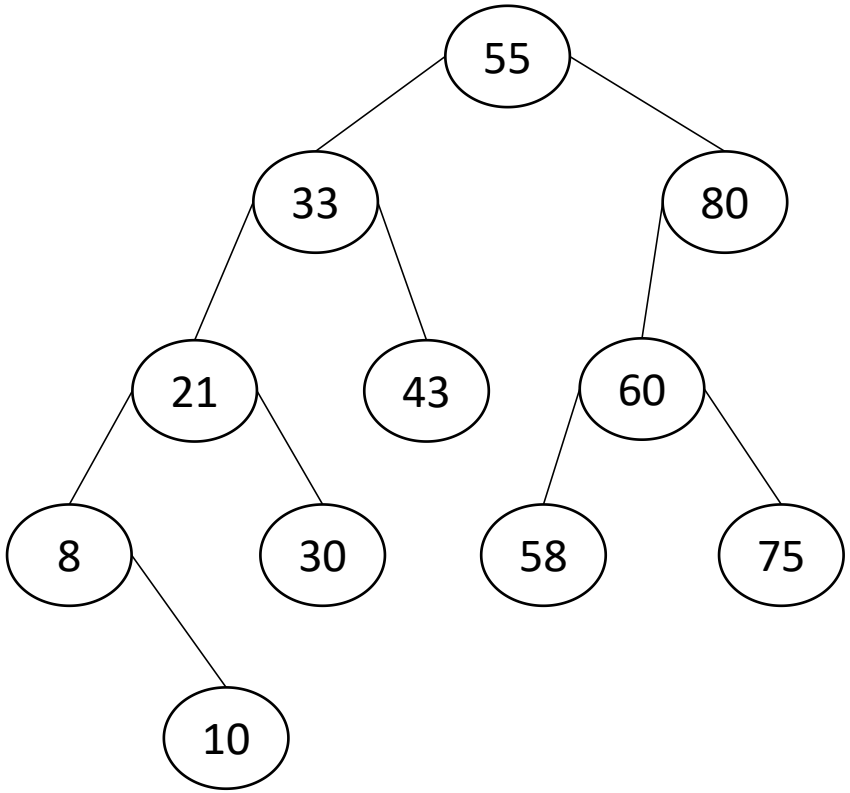
Complexity : O(Log2(n))

Binary trees

Binary search trees : Insertion

The inserted element is always a leaf

Inserting 10, then 58



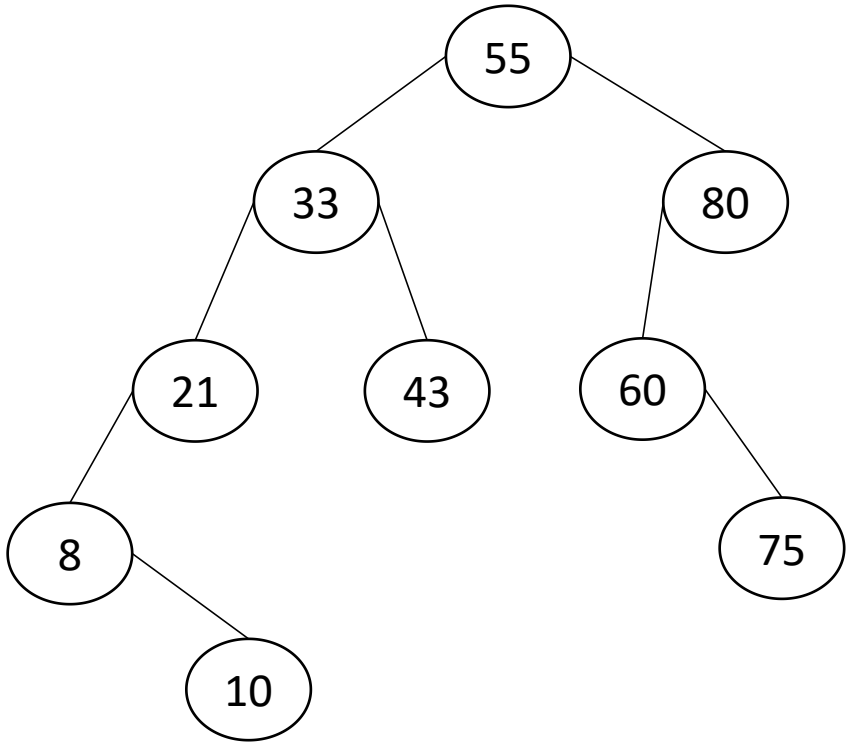
Complexité : O(1)

Binary trees

Binary search trees : Deletion

Several cases to consider
let n be the node to delete

LC(n) = nil & RC(n)= Nil	Replace n with nil
LC(n) = nil & RC(n) # Nil	Replace n par RC(n)
LC(n) # nil & RC(n)= Nil	Replace n par LC(n)
LC(n) # nil & RC(n) # Nil	let p be the inorder successor of the node n - Replace Node_value(n) par Node_value(p) - Replace p par RC(p)



Complexity : O(Log2(n))

Binary trees

Application 1 : Search for Duplicates in a Sequence of Numbers

Input : a sequence of numbers to read
Output : Display the doubles

To detect duplicates, we must save the numbers already read

Problem : Where save them ?

Pseudo algorithm:
Let S be the chosen data structure
For each number n read :
    Search for n in S
    If n exists
        Write (n)
    Else
        Insert(n) in S
    Endif
Enfor

Binary trees

Application 1 : Search for Duplicates in a Sequence of Numbers

Solution 1 : Array
- not ordered :
  - Linear search : O(n)
  - Insertion at the end : O(1)
- ordered :
  - Binary search : O(Log2(n))
  - Insertion with shifting : O(n)

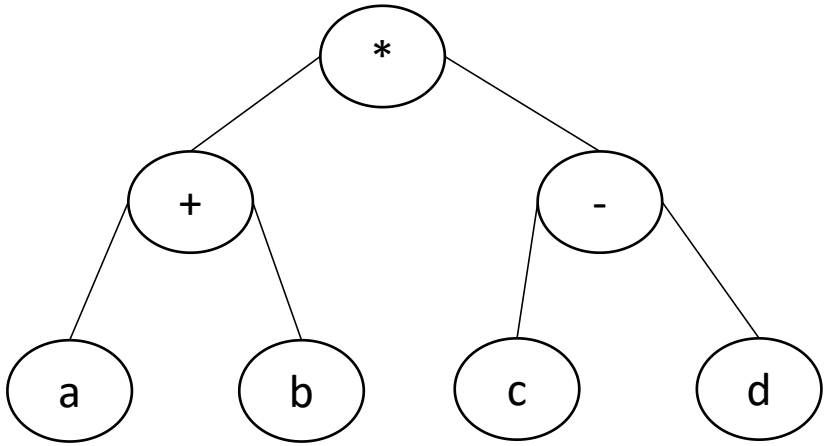
Solution 2 : Linked list
- not ordered :
  - Linear search : O(n)
  - Insertion at the end : O(1)
- ordered :
  - Linear search : O(n)
  - Insertion : O(1)

Solution 3 : Binary search tree
- Binary search : O(Log2(n))
- Insertion O(1)

Binary trees

Application 2 : Arithmetic expression

An arithmetic expression can be represented by a binary tree:
Operators : internal nodes
Operands : External nodes



(a + b) * (c - d)

Preorder traversal (nT1T2) : * + a b - c b (prefix polish notation)

Postorder traversal (T1T2n) : a b + c d - * (postfix polish notation)

Inorder traversal (T1nT2) : a + b * c - d (infix polish notation)

Binary trees

Application 2 : Arithmetic expression (algorithm)

Let A be the binary tree representing an arithmetic expression

The algorithm uses a postorder traversal

Oper(Op, A, B) : perform the operation Op on two values A and B

```
Eval(A) :
IF Leaf(A) :
        Eval := Node_value(A)
   ELSE
       Eval := Oper(Node_value(A), Eval(LC(A), Eval(RC(A))
ENDIF
```

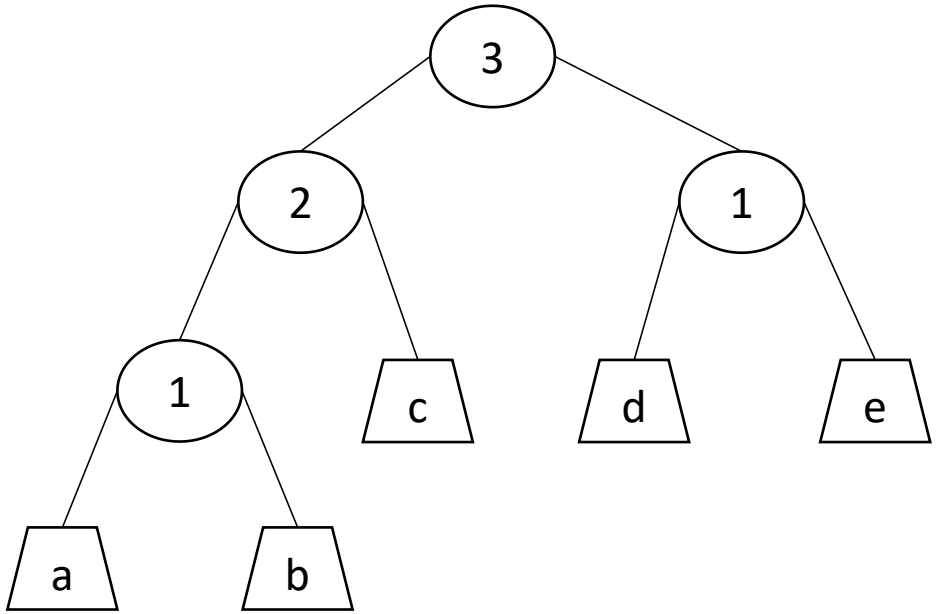
Complexity : O(n)

Binary trees

Application 3 : Representing Linked Lists by Binary Trees

A linked list can be represented by a binary tree:
Leaves: Items of the linked list
Internal nodes : Number of leaves in the left subtree

Goal: Quickly find the k-th element

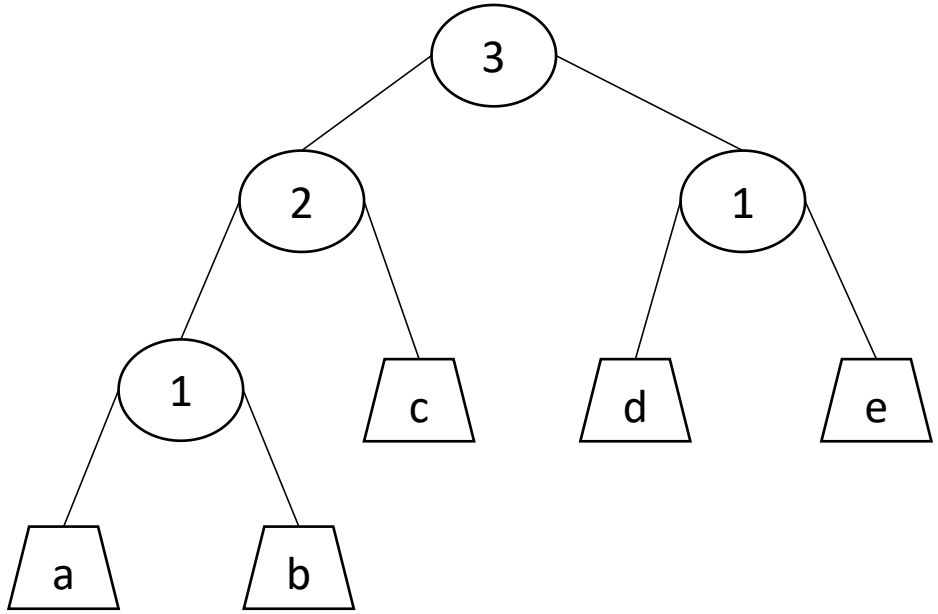


Binary trees

Application 3 : Representing Linked Lists by Binary Trees

Algorithm : Searching for the k-th element

```
R := K
P := Tree
WHILE NON Leaf(P) :
      IF R <= Count(P) :
           P := LC(P)
      ELSE
           R := R - Count(P)
           P := RC(P)
      ENDIF
ENDWHILE
```



Leaf(P) : True if P is a leaf, false otherwise

Count(P) : Number of leaves in the left subtree of the node P

Binary trees

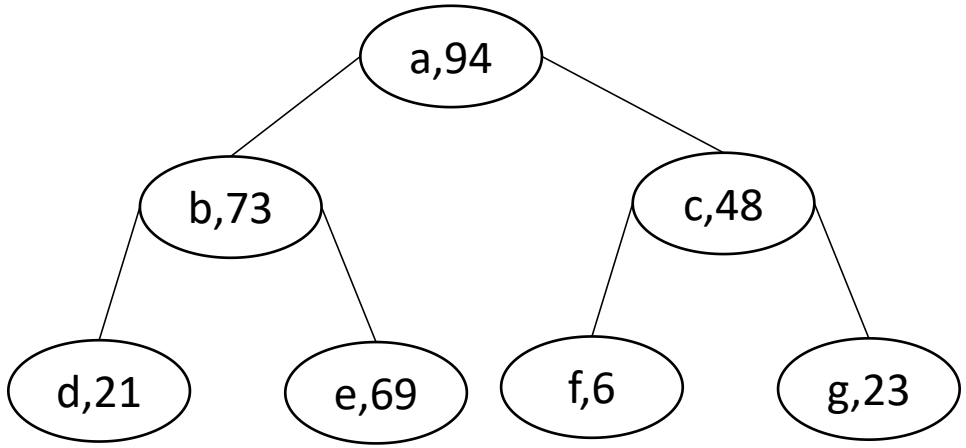
Application 4 : Priority queue

The queue is represented by a special binay tree (Heap)
Node = (Information + Priority)

All the levels of the tree are full (Except possibly the last)

The root holds the element with the largest priority

Each non-leaf node has a priority greater then its children



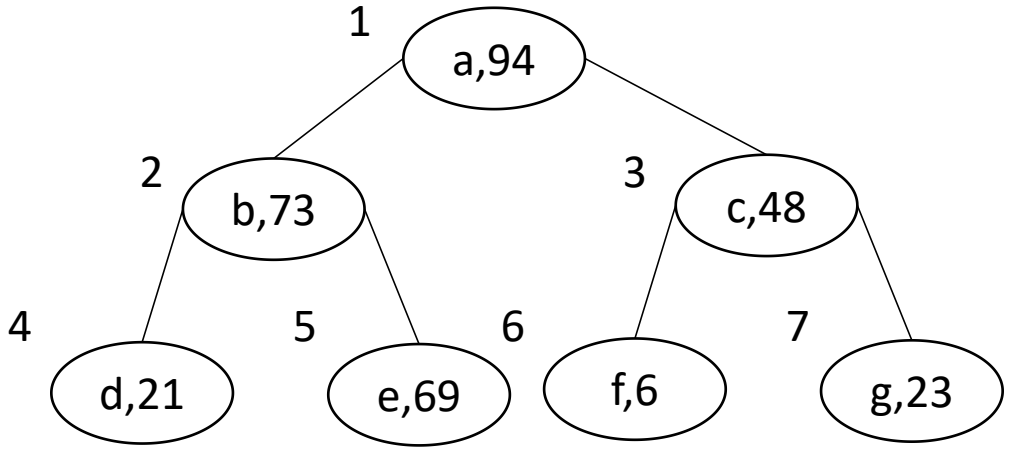
Binary trees

Application 4 : Priority queue

Nodes are numbered: the node at position P is the parent of nodes at positions 2P and 2P+1

To go to node at position P
P = $b_1b_2b3....b_n$. (Binary Representation)
For i=2, n
  If b_i = 0 Go to left Else Go to the right Endif
Endfor

To go to node at position 6
6 = **1 1 0**
From the root, go to the right, then to left



Binary trees

Application 4 : Priority queue (Operations)

Enqueue(V) =

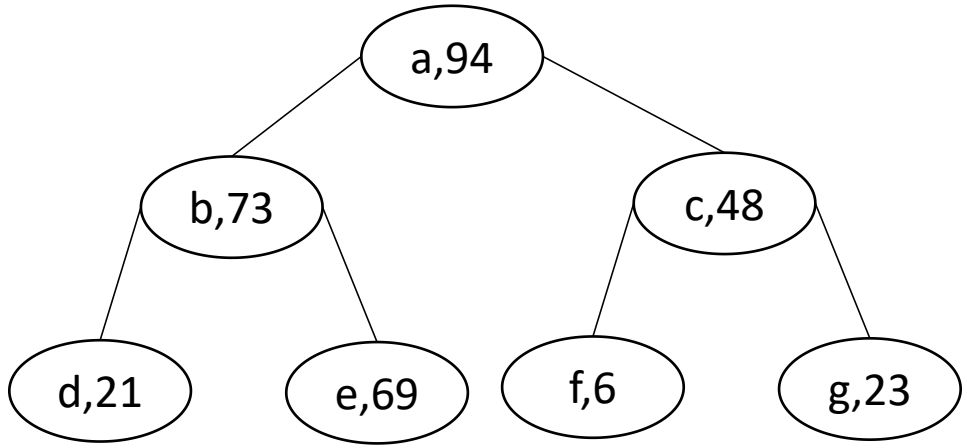
- Add a node with value V to the last level, the rightmost
- Swap with its ascendants until its priority is lower than that of its parent

→ O(log n)

Dequeue(V) =

- Get in V the root value and replace it with the rightmost node of the last level
- Swap with its descendants until its priority is greater than that of its children

→ O(log n)



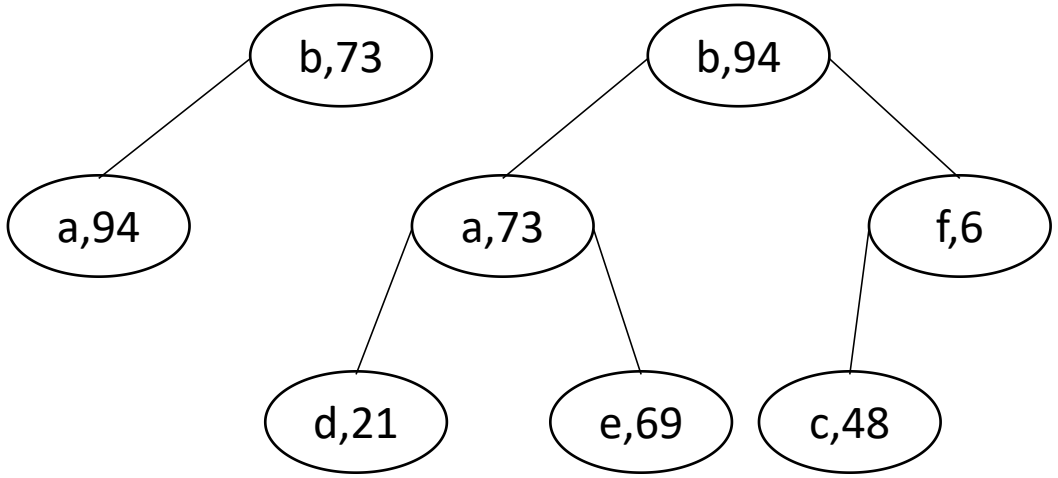
Better than an implementation with an ordered linked list

Binary trees

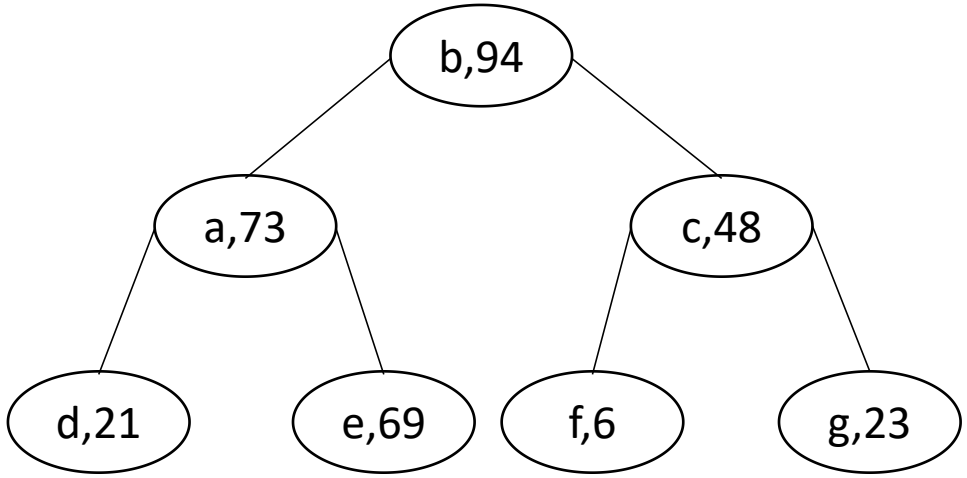
Application 4 : Priority queue (Example : Enqueuing)

b,73 a,94 f,6 d,21 e,69 c,48 g,23

Nb_heap	1	2	3	4	5	6	7
Binary		10	11	100	101	110	111



Permutation



Permutation

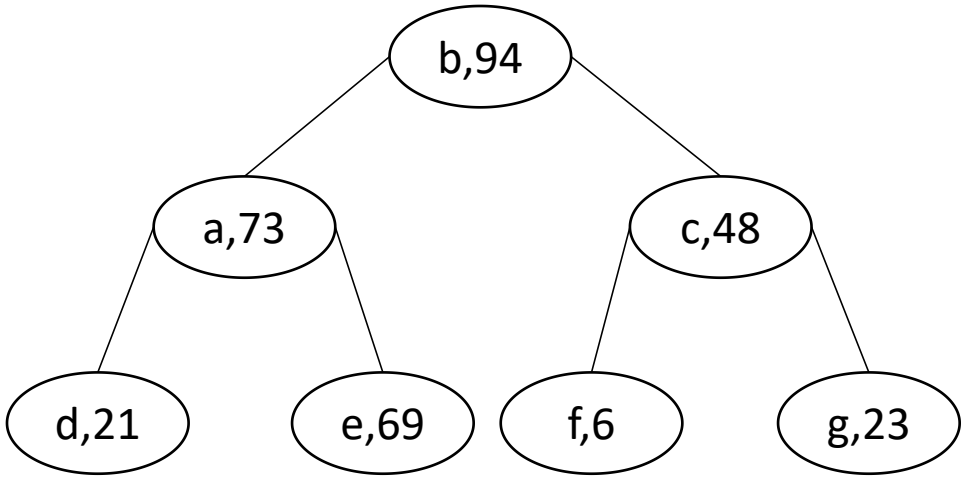
Binary trees

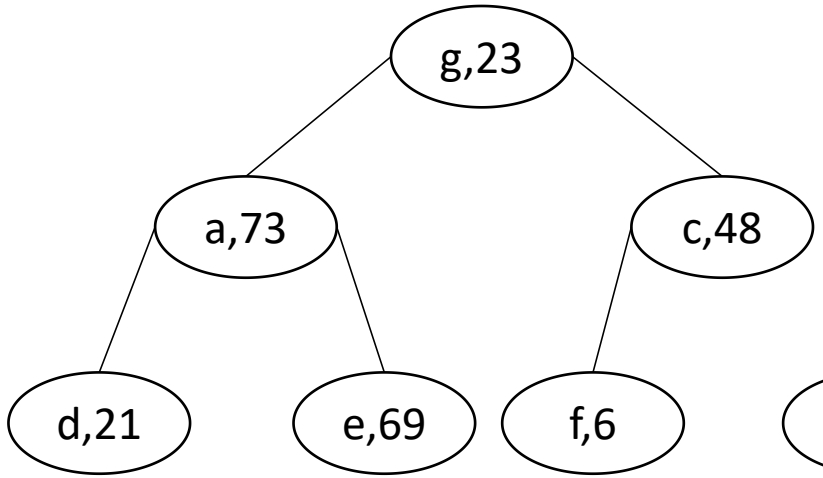
Application 4 : Priority queue (Example : Dequeuing)

Nb_heap 7

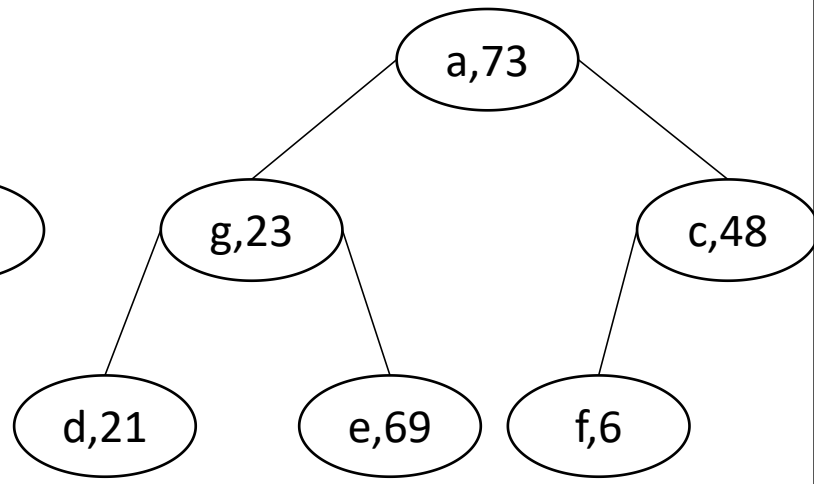
Binary 111

b





Permutation



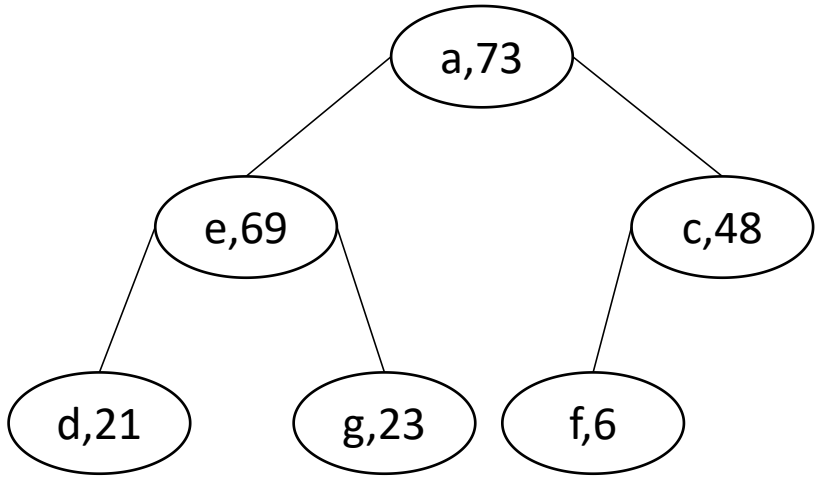
Permutation

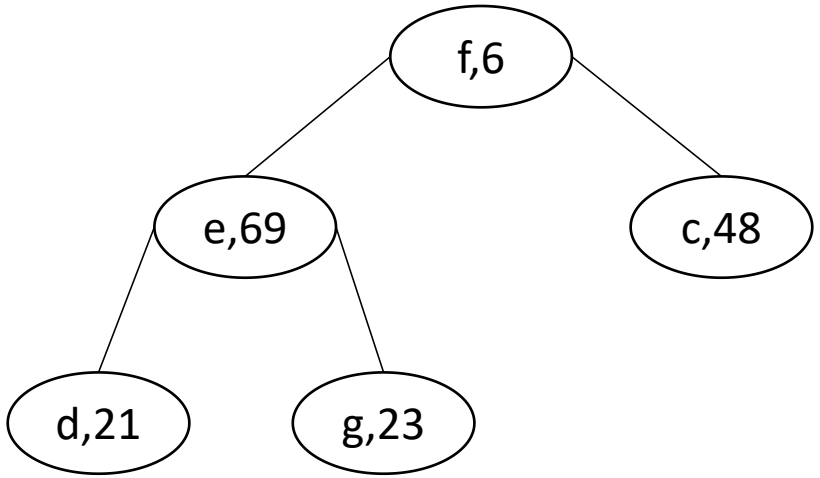
Binary trees

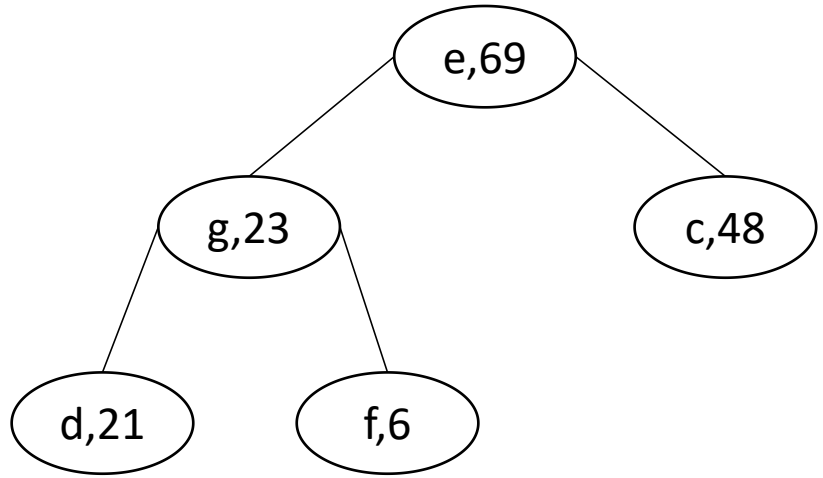
Application 4 : Priority queue (Example : Dequeuing)

Nb_heap	7	6
Binary	111	110

b a







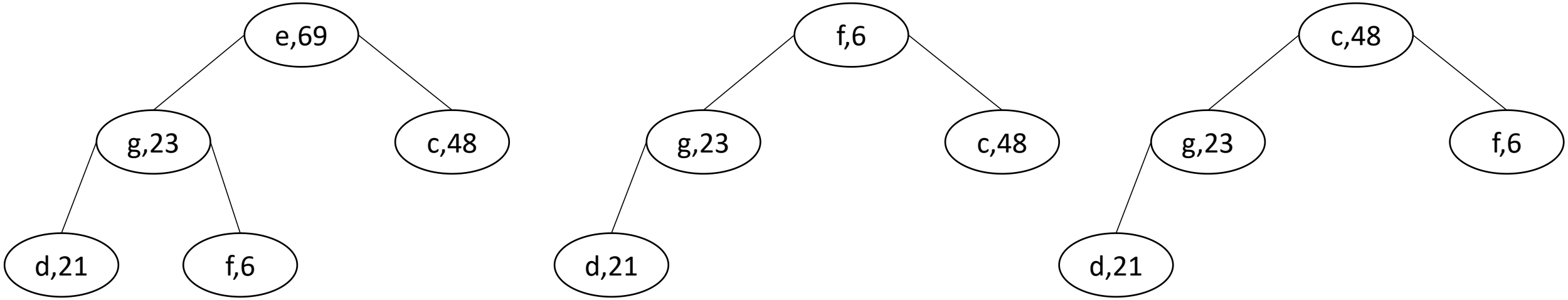
2 permutations

Binary trees

Application 4 : Priority queue (Example : Dequeuing)

Nb_heap	7	6	5
Binary	111	110	101

b a e



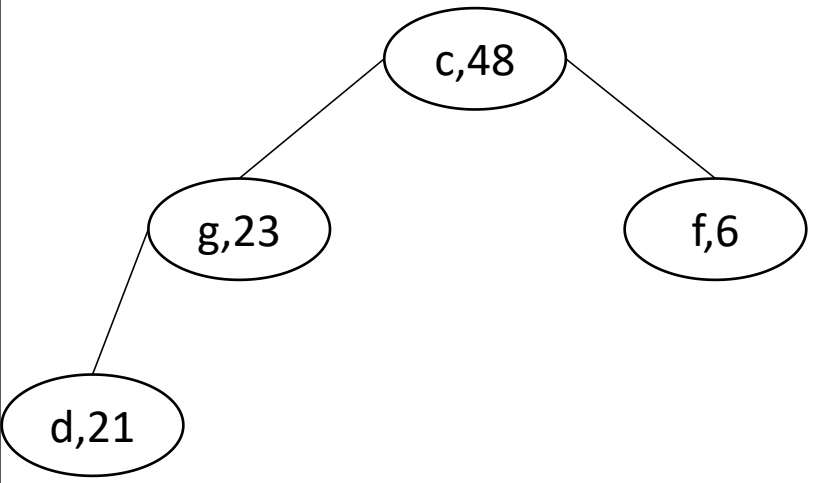
Permutation

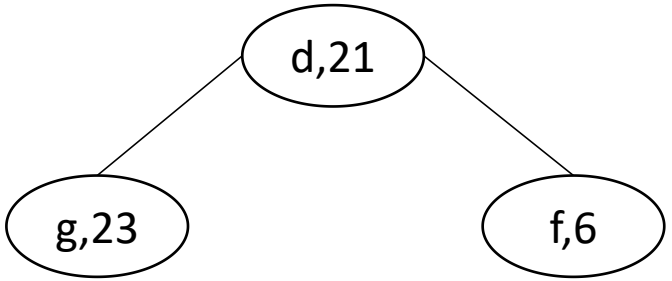
Binary trees

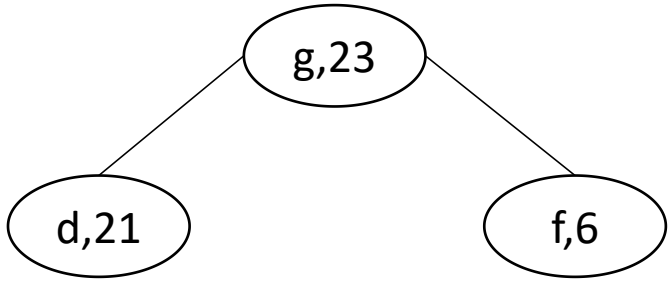
Application 4 : Priority queue (Example : Dequeuing)

Nb_heap	7	6	5	4
Binary	111	110	101	100

b a e c







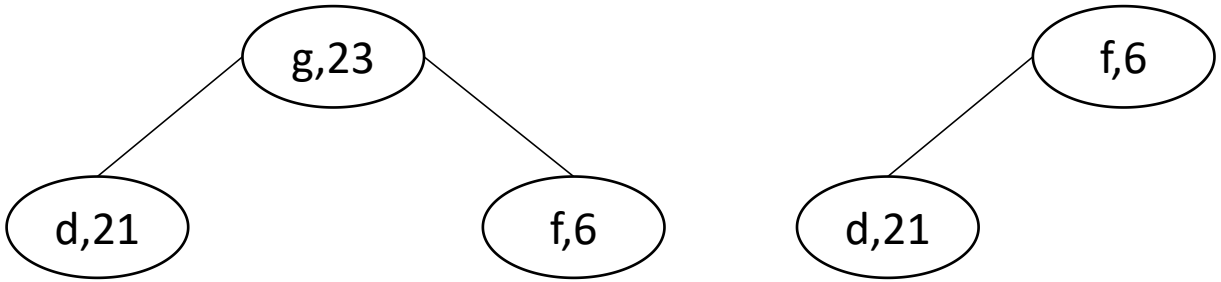
Permutation

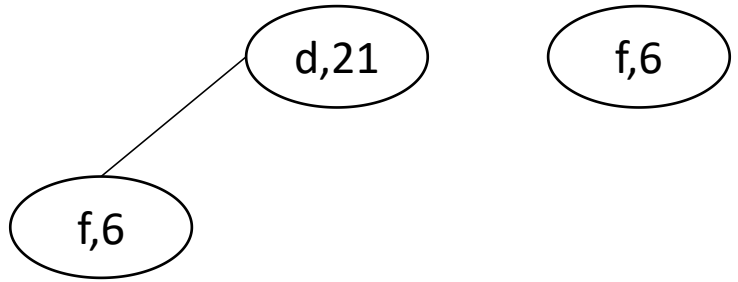
Binary trees

Application 4 : Priority queue (Example : Dequeuing)

Nb_heap	7	6	5	4	3	2	1	0
Binary	111	110	101	100	11	10		

b a e c g d f

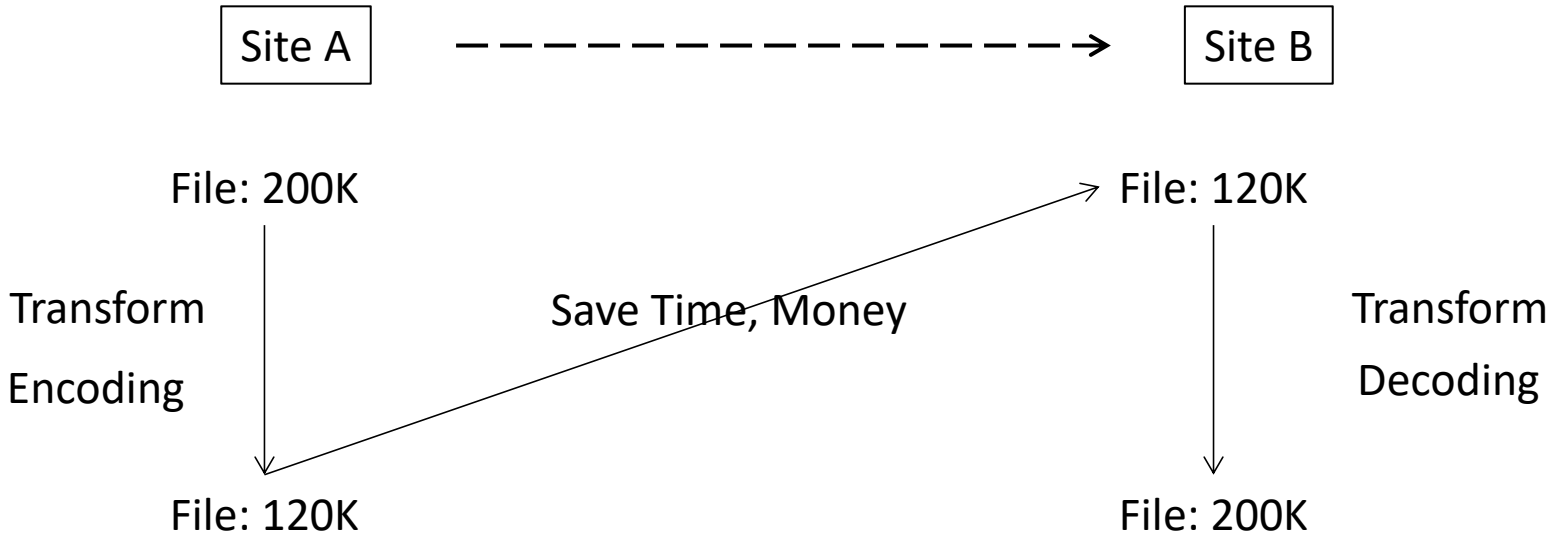




Permutation

Binary trees

Application 5 : Huffman's Code



Application : Data Compression/Decompression (Winzip, Rar,..)

Binary trees

Application 5 : Huffman's Code

Let A be an alphabet with the characters a, b, c, d, e appearing with given probabilities

Character	Probability
a	0.12
b	0.4
c	0.15
d	0.08
e	0.25

Encode each character using a sequence of 0s and 1s, ensuring that no code is a prefix of another code.

The prefix property ensures the decoding operation

Binary trees

Application 5 : Huffman's Code

The 2 codes have the prefix property.

For code 2, the sequence 1101001 is decoded to bcd.

Decoding principle:
1. Successively take a sequence of 1 bit, 2 bits, ... and see if it corresponds to a code.
2. If Yes, decode it and remove it from the sequence.

Repeat 1 et 2 until the sequence becomes empty

Character	Probability	Code1	Code2
a	0.12	000	000
b	0.4	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

1 1 0 1 0 0 1

b c d

Scenario: decoding (Code 2)

Binary trees

Application 5 : Huffman's Code

How to compar the 2 codes ?

Calculation of the average length

Code 1 :

Average length = 3

Code 2 :

Average length = 2.22

Average length = Sum (L_i*P_i), i=1, 5

L_i : code length of the i-th character

P_i : probability of the i-th character

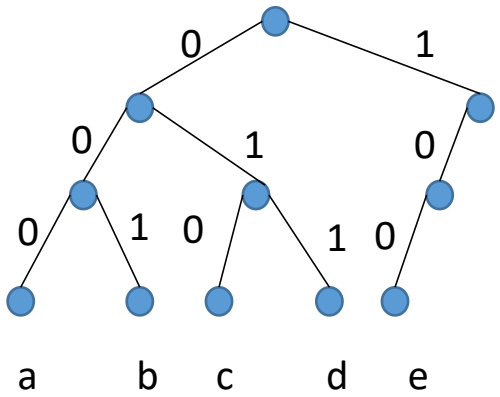
Character	Probability	Code1	Code2
a	0.12	000	000
b	0.4	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

How to find the best code ?

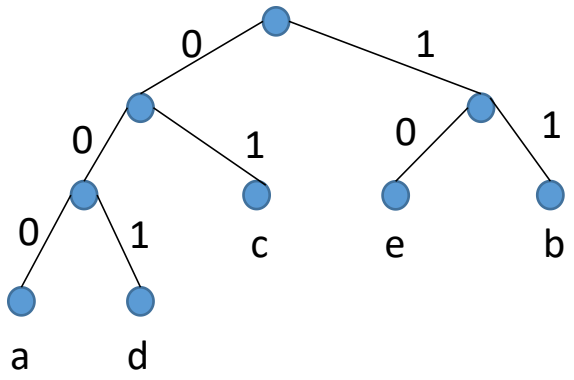
Binary trees

Application 5 : Huffman's Code

To each code, we can associate a binary tree



Code 1



Code 2

Character	Probability	Code1	Code2
a	0.12	000	000
b	0.4	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

Conversely, we can draw a binary tree with exactly 5 leaves and associate it a code.

Binary trees

Application 5 : Huffman's Code

Technique :

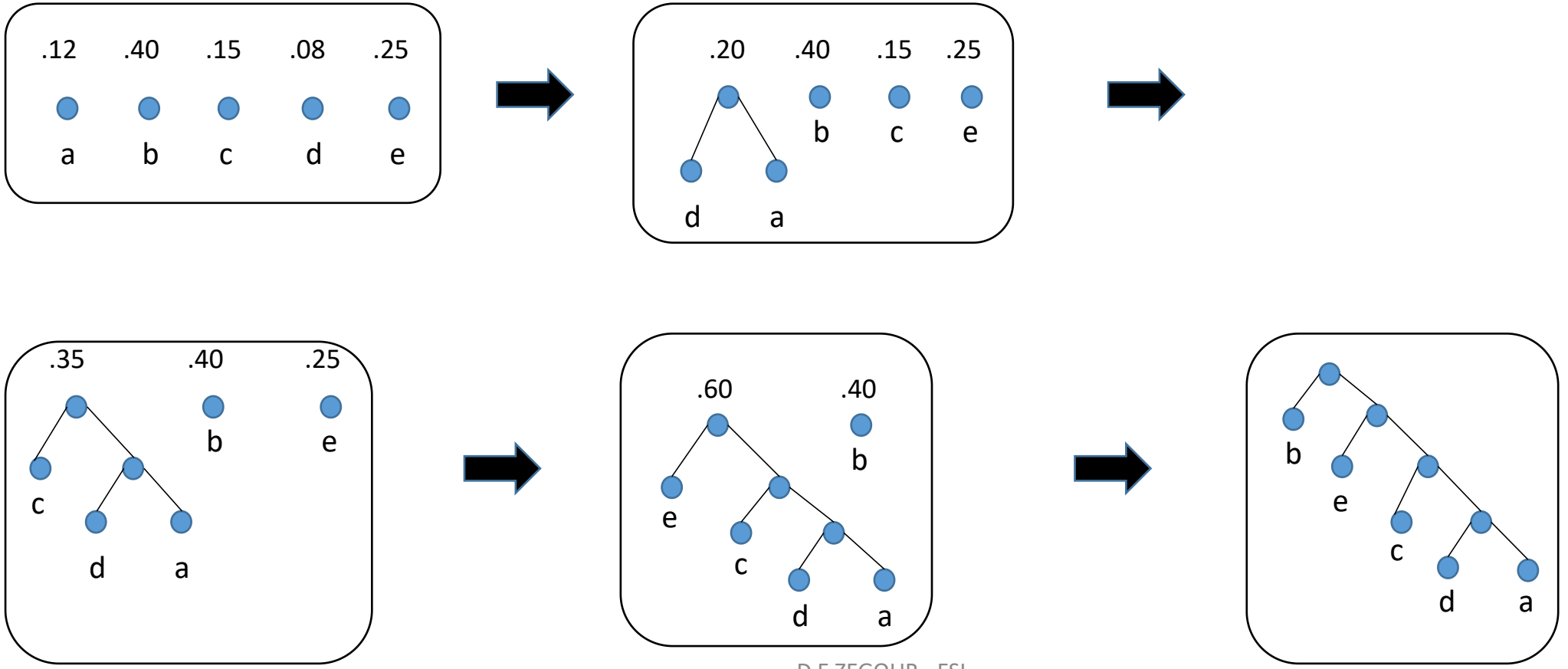
1. Select two characters, a and b, with the lowest probabilities.

2. Replace them with a fictitious character x such that p(x) = p(a) + p(b).

3. Repeat steps 1 and 2 recursively.

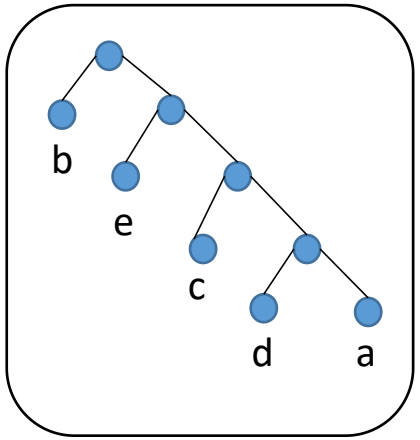
Binary trees

Application 5 : Huffman's Code



Binary trees

Application 5 : Huffman's Code



Character	Probability	Code
a	0.12	1111
b	0.4	0
c	0.15	110
d	0.08	1110
e	0.25	10

Average length : 2.17

Binary trees

C Dynamic Implementation

```
#include <stdio.h>
#include <stdlib.h>
/** -Implementation- **\: BINARY SEARCH TREE OF INTEGERS**/

/** Binary search trees **/
typedef int Typeelem_Ai   ;
typedef struct Node_Ai * Pointer_Ai ;
struct Node_Ai
 {
   Typeelem_Ai  Val ;
   Pointer_Ai Lc ;
   Pointer_Ai Rc ;
   Pointer_Ai Parent ;
  } ;

Typeelem_Ai Node_value_Ai( Pointer_Ai P )
 { return P->Val;   }

Pointer_Ai Lc_Ai( Pointer_Ai P)
 { return P->Lc ; }
```

```
Pointer_Ai Rc_Ai( Pointer_Ai P)
 { return P->Rc ; }
Pointer_Ai Parent_Ai( Pointer_Ai P)
 { return P->Parent ; }
void Ass_node_val_Ai ( Pointer_Ai P, Typeelem_Ai Val)
 {
    P->Val = Val ;
 }
void Ass_lc_Ai( Pointer_Ai P, Pointer_Ai Q)
 { P->Lc =  Q;  }
void Ass_rc_Ai( Pointer_Ai P, Pointer_Ai Q)
 { P->Rc =  Q ; }
void Ass_parent_Ai( Pointer_Ai P, Pointer_Ai Q)
 { P->Parent =  Q ; }
void Allocate_node_Ai( Pointer_Ai *P)
 {
   *P = (struct Node_Ai *) malloc( sizeof( struct Node_Ai))   ;
   (*P)->Lc = NULL;
   (*P)->Rc = NULL;
   (*P)->Parent = NULL;
 }
```

Binary trees

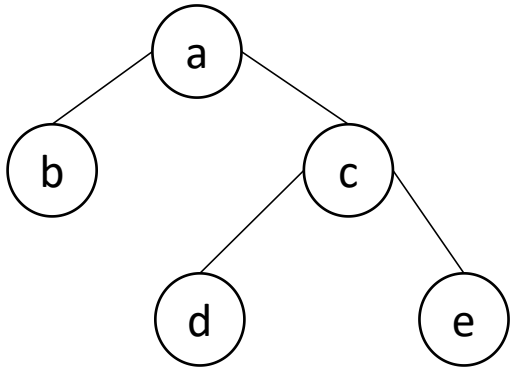
C Dynamic Implementation

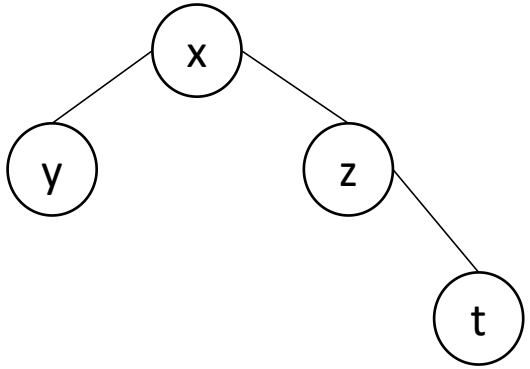
```
 void Free_node_Ai( Pointer_Ai P)
  { free( P ) ; }

 /** Variables of main program **/
 Pointer_Ai B=NULL;
 int main(int argc, char *argv[])
  {
    system("PAUSE");
    return 0;
}
```

Binary trees

Static Implementation





Several binary trees in the same array

0	3	a	4	V
1	-1	y	-1	V
2	1	x	5	V
3	-1	b	-1	V
4	6	c	8	V
5	-1	z	7	V
6	-1	d	-1	V
7	-1	t	-1	V
8	-1	e	-1	V
				F
				...
Max				F

Binary trees

Static Implementation

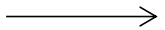
An element = (Value, RC, LC, Occupied).

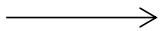
The field "Occupied" is necessary for the operations Allocate_node and Free_node

An initialization phase is mandatory before using this array (Occupied field set to False).

So, the array is global.

A binary tree is defined by the index of its first element.





0	3	a	4	V
1	-1	y	-1	V
2	1	x	5	V
3	-1	b	-1	V
4	6	c	8	V
5	-1	z	7	V
6	-1	d	-1	V
7	-1	t	-1	V
8	-1	e	-1	V
9				F
				...
Max				F

Binary trees

C Static Implementation

```
#define Max 100
#define True 1
#define False 0
#define Nil -1
typedef int Bool;
typedef int Anytype;
struct Typebst
 {
  Anytype Info ;
  int Lc;
  int Rc;
  Bool Occupied;
 };
struct Typebst Bst[Max];
/* Initialisation */
void Init()
 {
  int I;
  for (I=0; I<Max; I++)
    Bst[I].Occupied = False;
 }
```

```
void Allocate_node ( int *I )
 {
  Bool Found;
  *I = 0;
  Found= False;
  while ( *I < Max && !Found)
   if ( Bst[*I].Occupied )
     *I++  ;
   else
     Found= True;
  if  ( !Found)  *I = -1;
 }
void Free_node ( int I )
 {
  Bst[I].Occupied = False ;
 }
Anytype Node_value ( int I )
 {
  return( Bst[I].Info );
 }
```

Binary trees

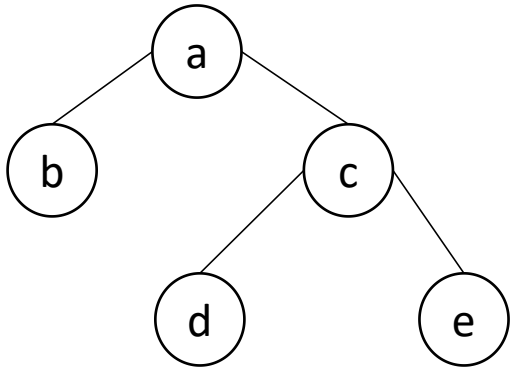
C Static Implementation

```
int Rc ( int I )
 {
  return ( Bst[I].Rc ) ;
 }
int Lc ( int I )
 {
  return ( Bst[I].Lc ) ;
 }
void Ass_node_val ( int I, Anytype Val)
 {
  Bst[I].Info = Val;
 }
void Ass_Rc ( int I, int J)
 {
  Bst[I].Rc = J;
 }
void Ass_Lc ( int I, int J)
 {
  Bst[I].Lc = J;
 }
int main(int argc, char *argv[])
 {
  system("PAUSE");
  return 0;
 }
```

```
int main(int argc, char *argv[])
 {
  system("PAUSE");
  return 0;
 }
```

Binary trees

Implementation (Sequential Representation)



Idea : Do not represent the indices of children

a	b	c			d	e	
1	2	3	4	5	6	7	Max

The root is at position 1

the node at position p is the parent of nodes at positions 2p (left child) and 2p+1 (Right child).

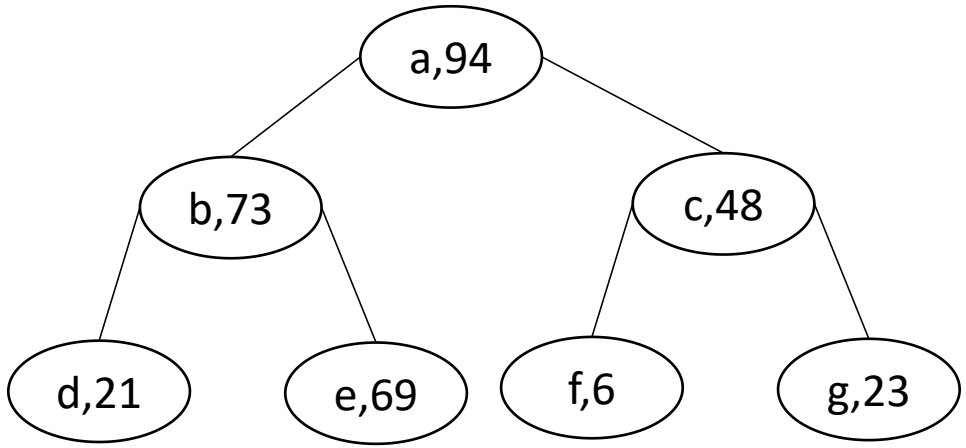
If a left child is at position p, its right brother is at position p+1;

If a right child is at position p, its left brother is at position p-1.

Parent(p) : it’s the node at position p DIV 2.

Binary trees

Implementation (Sequential Representation : Application : Priority queue)



a,94	b,73	c,48	d,21	e,69	f,6	g,23	.....
1	2	3	4	5	6	7	Max

Enqueue(F, v) : n := n+ 1, T[n] := v ; Possible permutations
(Parent(P) : P Div 2)

Dequeue(F, v) : T[1] := T[n]; n:=n - 1 ; Possible permutations
(LC(P) =2P ; RC(P) = 2P + 1)