



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Ingegneria
“Enzo Ferrari”

Fondamenti di Informatica II

Alberi binari come ADT

L'ADT (ABSTRACT DATA TYPE) ALBERO BINARIO

In generale, un tipo di dato astratto T è definito come:

- Un dominio base D
- Un insieme di funzioni $F = \{F1, F2, \dots\}$ sul dominio D
- Un insieme di predicati $P = \{P1, P2, \dots\}$ sul dominio D

$$T = \{D, F, P\}$$

Un **albero binario** è un tipo di dato astratto tale che:

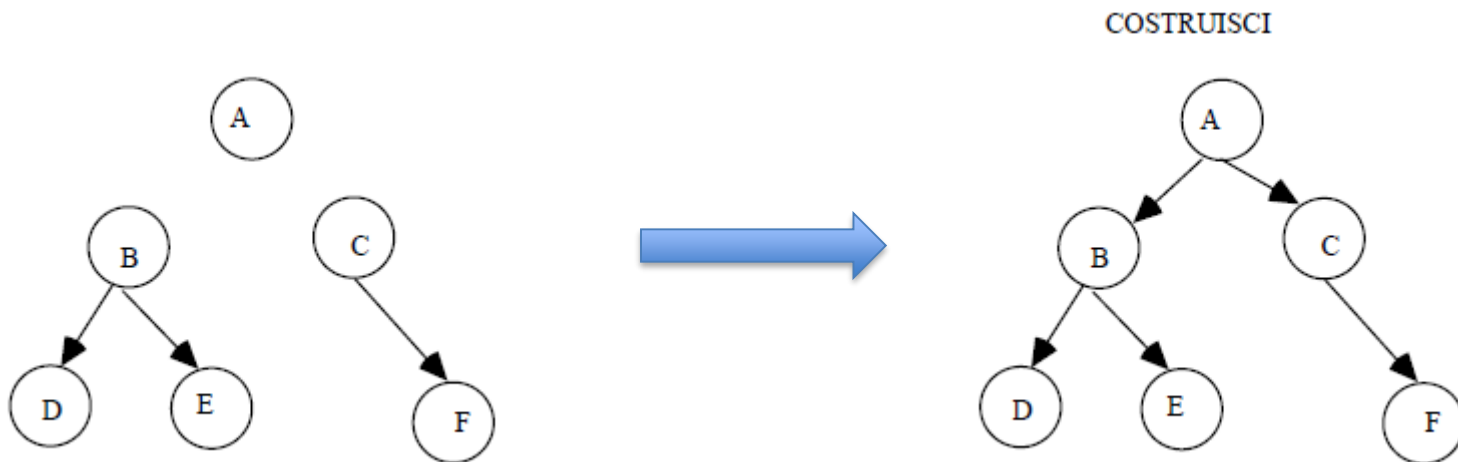
- D può essere qualunque
- $F = \{\text{consTree}, \text{root}, \text{left}, \text{right}, \text{tail}, \text{emptytree}\}$
 - $\text{consTree} : D \times \text{Tree} \rightarrow \text{Tree}$ (costruttore)
 - $\text{root} : \text{Tree} \rightarrow D$ (selettore 'radice')
 - $\text{left} : \text{Tree} \rightarrow \text{Tree}$ (selettore 'figlio sinistro')
 - $\text{right} : \text{Tree} \rightarrow \text{Tree}$ (selettore 'figlio destro')
 - $\text{emptylist} : \rightarrow \text{Tree}$ (creazione albero 'vuoto')
- $P = \{\text{isempty}\}$
 - $\text{isempty} : \text{Tree} \rightarrow \text{bool}$ (test di albero vuoto)

L'ADT ALBERO BINARIO

ESEMPIO: L'OPERAZIONE DI COSTRUZIONE (constree):

consTree costruisce un nuovo albero a partire da:

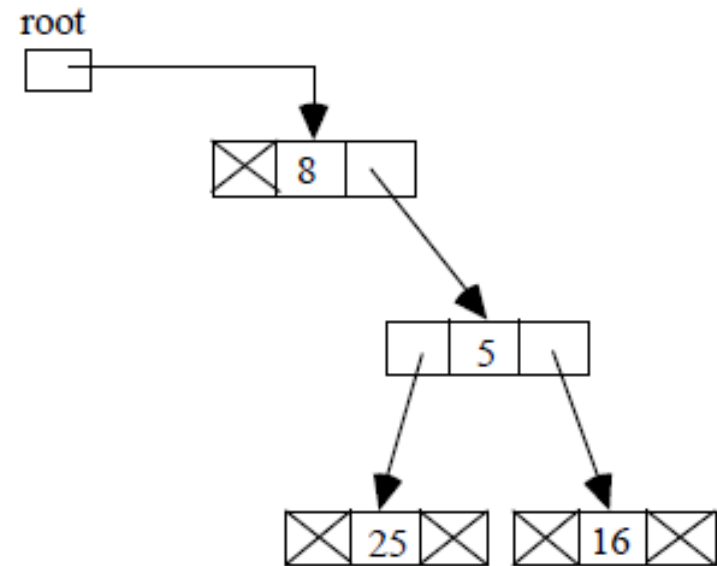
- un elemento (da inserire nel nuovo nodo radice)
- due alberi (da inserire rispettivamente come figlio sinistro e figlio destro del nuovo albero)



ALBERI BINARI: RAPPRESENTAZIONE

La rappresentazione più efficiente e diretta di un albero binario è attraverso strutture e puntatori

```
struct Node
{
    ElemType value;
    struct Node *left;
    struct Node *right;
};
typedef struct Node Node;
```



*Node verrà usato per riferirsi alla radice dell'Albero

ADT ALBERO BINARIO

```
struct Node
{
    ElemType value;
    struct Node *left; struct Node *right;
};
typedef struct Node Node;
// primitive
Node* CreateEmptyTree(void);
Node* CreateRootTree(const ElemType *e, Node* l, Node* r);
const ElemType* GetRootTree(const Node *n);
Node* LeftTree(const Node *n);
Node* RightTree(const Node *n);
void DeleteTree(Node *n);
bool IsEmptyTree(const Node *n);
bool IsLeafTree(const Node *n);
```

ADT ALBERO BINARIO

// non primitive

```
void WritePreOrderTree(const Node *n, FILE *f);  
void WriteStdoutPreOrderTree(const Node *n);  
void WriteInOrderTree(const Node *n, FILE *f);  
void WriteStdoutInOrderTree(const Node *n);  
void WritePostOrderTree(const Node *n, FILE *f);  
void WriteStdoutPostOrderTree(const Node *n);
```

ADT ALBERO BINARIO

```
Node* CreateEmptyTree(void) {
    return NULL;
}

Node* CreateRootTree(const ElemType *e, Node *l, Node *r) {
    Node* t = malloc(sizeof(Node));
    t->value = ElemCopy(e);
    t->left = l;
    t->right = r;
    return t;
}

const ElemType* GetRootTree(const Node *n){
    if (IsEmptyTree(n)) {
        printf("ERROR: Alla funzione 'GetRootTree()' e' stato passato
                un albero vuoto (NULL pointer).\n");

        exit(1); }
    else {
        return &n->value; }
}
```

ADT ALBERO BINARIO

```
Node* LeftTree(const Node *n) {
    if (IsEmptyTree(n)) {
        return NULL; }
    else {
        return n->left; }
}
Node* RightTree(const Node *n) {
    if (IsEmptyTree(n)) {
        return NULL; }
    else {
        return n->right; }
}
void DeleteTree(Node *n) {
    if (IsEmptyTree(n)) {
        return; }
    Node *l = LeftTree(n);
    Node *r = RightTree(n);
    ElemDelete(&n->value);
    free(n);
    DeleteTree(l); DeleteTree(r);
}
```


ADT ALBERO BINARIO

```
bool IsEmptyTree(const Node *n) {  
    return n == NULL; }  
  
bool IsLeafTree(const Node *n){  
    return LeftTree(n) == NULL  
        && RightTree(n) == NULL; }  
  
// non primitive  
static void WritePreOrderTreeRec(const Node *n, FILE *f) {  
    if (IsEmptyTree(n)) {  
        return; }  
    printf("\t"); WriteElem(GetRootTree(n), f);  
    WritePreOrderTreeRec(LeftTree(n), f);  
    WritePreOrderTreeRec(RightTree(n), f);  
}
```

ADT ALBERO BINARIO

```
static void WriteInOrderTreeRec(const Node *n, FILE *f) {
    if (IsEmptyTree(n)) {
        return; }
    WriteInOrderTreeRec(LeftTree(n), f);
    printf("\t"); WriteElem(GetRootTree(n), f);
    WriteInOrderTreeRec(RightTree(n), f);
}

static void WritePostOrderTreeRec(const Node *n, FILE *f) {
    if (IsEmptyTree(n)) {
        return; }
    WritePostOrderTreeRec(LeftTree(n), f);
    WritePostOrderTreeRec(RightTree(n), f);
    printf("\t"); WriteElem(GetRootTree(n), f);
}
```

COSTRUZIONE ALBERO BINARIO

```
// Per ogni nodo (elemento i-esimo del vettore) considero come figlio sinistro
// l'elemento del vettore di indice  $i * 2 + 1$ , e come figlio destro l'elemento
// di indice  $i * 2 + 2$ .

Node* CreateTreeFromVectorRec(const int *v, size_t v_size, int i) {
    if (i >= (int)v_size) {
        return NULL;    }

    Node *l = CreateTreeFromVectorRec(v, v_size, i * 2 + 1);
    Node *r = CreateTreeFromVectorRec(v, v_size, i * 2 + 2);
    return CreateRootTree(&v[i], l, r);
}

Node* CreateTreeFromVector(const int *v, size_t v_size) {
    return CreateTreeFromVectorRec(v, v_size, 0); }

int main(void){
    int v[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    size_t v_size = sizeof(v) / sizeof(int);
    Node *tree = CreateEmptyTree();
    tree = CreateTreeFromVector(v, v_size);
    DeleteTree(tree);
    return EXIT_SUCCESS;
}
```