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, …} sul dominio D
- Un insieme di predicati P = {P1, P2, …} sul dominio D

T = {D, F, P}

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

- D può essere qualunque
- F = {consTree, root, left, right, tail, emptytree}

consTree :	D x Tree -> Tree	(costruttore)
root:	Tree -> D	(selettore 'radice')
left:	Tree -> Tree	(selettore 'figlio sinistro')
right:	Tree -> Tree	(selettore 'figlio destro')
emptylist:	-> Tree	(creazione albero 'vuoto')

- P = {isempty}

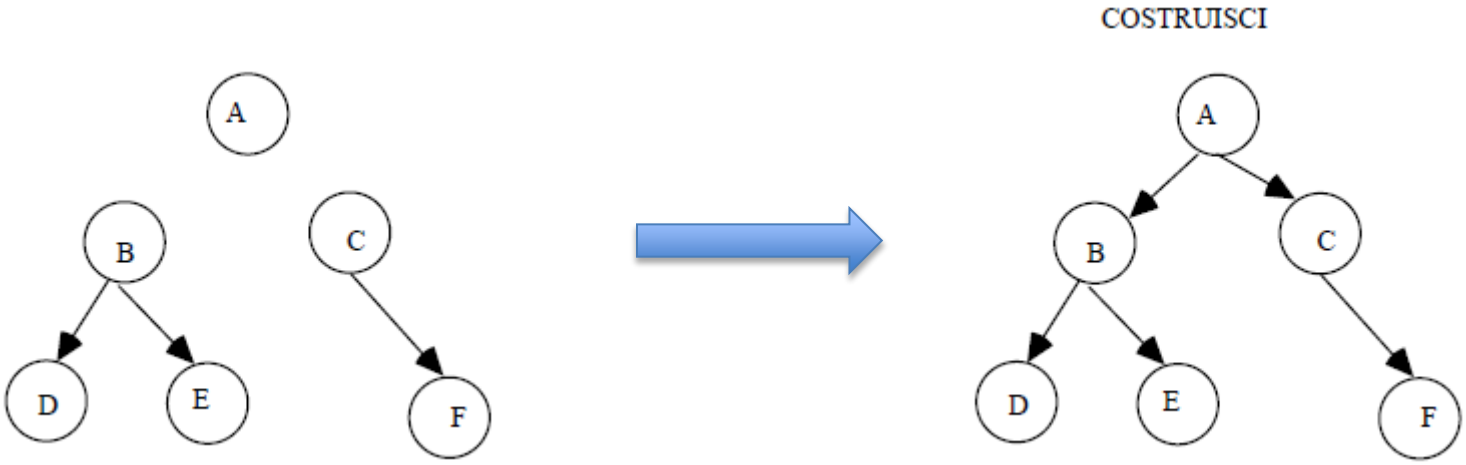
isempty	Tree -> bool	(test di albero vuoto)

L'ADT ALBERO BINARIO

ESEMPIO: L'OPERAZIONE DI COSTRUZIONE (constree):

consTree costruisce un nuovo albero a partire da:

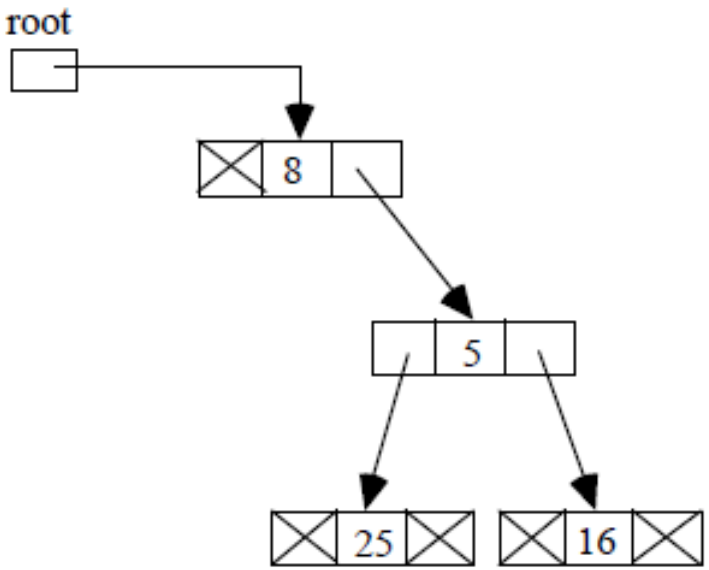
- un elemento (da inserire nel nuovo nodo radice)
- due alberi (da inserire rispettivamente come figlio sinistro e figlio destro del nuovo albero)



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 *TreeCreateEmpty (void);
Node *TreeCreateRoot(const ElemType *e, Node *l, Node *r);
const ElemType *TreeGetRootValue(const Node *n);
Node *TreeLeft(const Node *n);
Node *TreeRight(const Node *n);
void TreeDelete(Node *n);
bool TreeIsEmpty(const Node *n);
bool TreeIsLeaf(const Node *n);
```

ADT ALBERO BINARIO

```
// non primitive
void TreeWritePreOrder(const Node *n, FILE *f);
void TreeWriteStdoutPreOrder(const Node *n);
void TreeWriteInOrder(const Node *n, FILE *f);
void TreeWriteStdoutInOrder(const Node *n);
void TreeWritePostOrder(const Node *n, FILE *f);
void TreeWriteStdoutPostOrder(const Node *n);
```

ADT ALBERO BINARIO

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

Node *TreeCreateRoot(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 *TreeGetRootValue(const Node *n){
    if (TreeIsEmpty(n)) {
        printf("ERROR: Alla funzione 'TreeGetRootValue()' e' stato passato
                                        un albero vuoto (NULL pointer).\n");
        exit(1); }
    else {
        return &n->value; }
}
```

ADT ALBERO BINARIO

```
Node *TreeLeft(const Node *n) {
    if (TreeIsEmpty(n)) {
        return NULL; }
    else {
        return n->left; }
}
Node *TreeRightTree(const Node *n) {
    if (TreeIsEmpty(n)) {
        return NULL; }
    else {
        return n->right; }
}
void TreeDelete(Node *n) {
    if (TreeIsEmpty (n)) {
        return; }
    Node *l = TreeLeft(n);
    Node *r = TreeRight (n);
    ElemDelete(&n->value);
    free(n);
    TreeDelete (l); TreeDelete(r);
}
```

ADT ALBERO BINARIO

```
bool TreeIsEmpty (const Node *n) {
    return n == NULL; }
bool TreeIsLeaf (const Node *n) {
    return TreeLeft(n) == NULL && TreeRight(n) == NULL;
}
// non primitive
static void TreeWritePreOrderRec(const Node *n, FILE *f) {
    if (TreeIsEmpty(n)) {
        return; }
    fprintf(f, "\t"); ElemWrite(TreeGetRootValue(n), f);
    TreeWritePreOrderRec(TreeLeft(n), f);
    TreeWritePreOrderRec(TreeRight(n), f);
}
```

ADT ALBERO BINARIO

```
static void TreeWriteInOrderRec(const Node *n, FILE *f) {
    if (TreeIsEmpty(n)) {
        return; }
    TreeWriteInOrderRec(TreeLeft(n), f);
    fprintf(f, "\t"); ElemWrite(TreeGetRootValue(n), f);
    TreeWriteInOrderRec(TreeRight(n), f);
}
static void TreeWritePostOrderRec(const Node *n, FILE *f) {
    if (TreeIsEmpty(n)) {
        return; }
    TreeWritePostOrderRec(TreeLeft(n), f);
    TreeWritePostOrderRec(TreeRight(n), f);
    fprintf(f, "\t"); ElemWrite(TreeGetRootValue(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 TreeCreateRoot(&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 = TreeCreateEmpty();
    tree = CreateTreeFromVector(v, v_size);
    TreeDelete(tree);
    return EXIT_SUCCESS;
}
```