57. Alberi binari
Gli alberi sono una struttura matematica che gioca un ruolo molto importante nella progettazione e nell'analisi di algoritmi: sono spesso utilizzati per descrivere proprietà dinamiche degli algoritmi.

Spesso utilizziamo strutture dati che rappresentano implementazioni concrete di alberi. Questo tipo di ADT lo incontriamo nella vita di tutti i giorni:

- l'albero genealogico della propria famiglia (da cui deriva la maggior parte della terminologia impiegata nella teoria degli alberi);
- nei tornei sportivi;
- per rappresentare l'organigramma di aziende;
- per rappresentare l'analisi sintattica dei linguaggi di programmazione;
- il file system di un sistema operativo;
- gerarchie

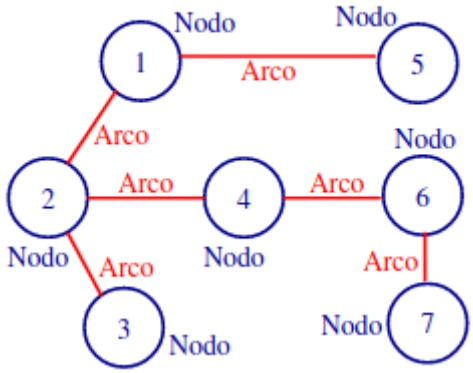
57.1 Tipi di alberi.
Esistono diversi tipi di alberi, ed è importante distinguere tra modello astratto e modello concreto (ovvero tra modello matematico e implementazione).
In ordine di generalità decrescente distinguiamo:

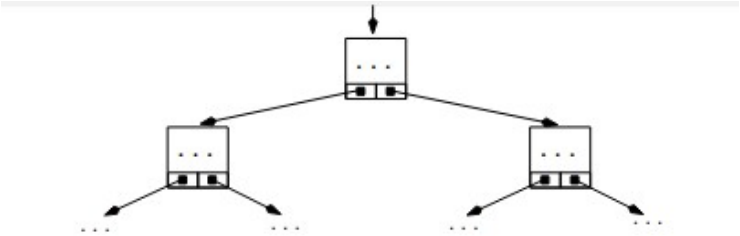
- Alberi generici.
- Alberi con radice.
- Alberi ordinati.
- Alberi M-ari.
- Alberi binari come caso particolare di albero M-ario.

57.2 Alberi (Teoria dei grafi).
In teoria dei grafi un albero è un grafo non orientato, nel quale due vertici qualsiasi sono connessi da uno e un solo cammino.

Definizione
Un albero è un insieme non vuoto di vertici ed archi (grafo) che soddisfa alcune proprietà:

- Un vertice (o nodo) è un oggetto semplice che può essere dotato di un nome, e di una informazione associata (denominata spesso chiave o key).
- Un arco è una connessione tra due nodi.
- Un grafo non orientato, connesso e privo di cicli.





57.3 Alberi binari
(Nella sua versione più semplice [figura superiore destra]) una albero binario *t* di oggetti di tipo *T* è definito come segue:

- *t* è un puntatore *NULL* (albero vuoto) oppure
- *t* è un puntatore ad un *nodo* (***struct***) contenente:
  - un campo *value* di tipo *T*
  - due campi *left* e *right* di tipo *albero* (puntatore a *nodo*, cioè: *nodo**)

Struttura che definisce gli elementi dell'albero.
struct node;
*typedef node * albero;*

struct node { T value;
albero left, right;
};

N.B.! Un albero è una struttura dati **dinamica**.

57.3.1 Alberi binari: terminologia.
I sottoalberi di un nodo *N* sono detti *sottoalbero sinistro* e *sottoalbero destro* di *N*.

Se un nodo *N* punta nell'ordine a due (eventuali) nodi *N1*, *N2:*

- *N1* e *N2* sono detti rispettivamente *figlio sinistro* e *figlio destro* di *N*
- *N* è detto nodo *padre* di *N1* e *N2*

In un albero binario ci possono essere tre tipi di nodi:

- Il *nodo radice*, che non ha padre.
- I *nodi foglia*, che non hanno figli.
- I *nodi intermedi*, che hanno padre e almeno un figlio.

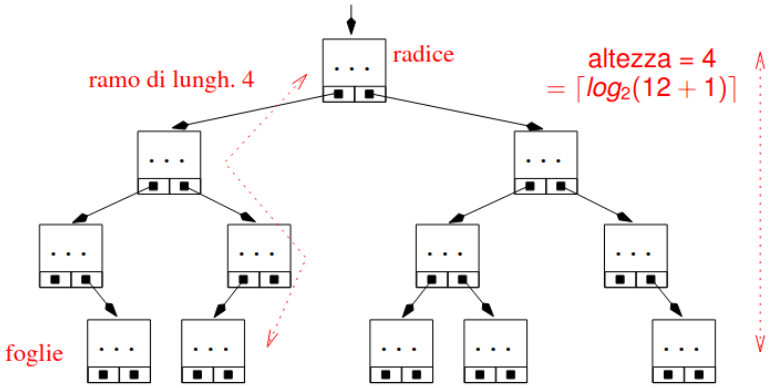
Una catena di nodi dalla radice a una foglia è detta *ramo.*
Il numero di nodi in un ramo è detto *lunghezza del ramo.*
La massima lunghezza di un ramo è detta *altezza dell'albero*.

L'altezza di un albero binario di *N* elementi è $h \in [log2(N + 1), N]$.

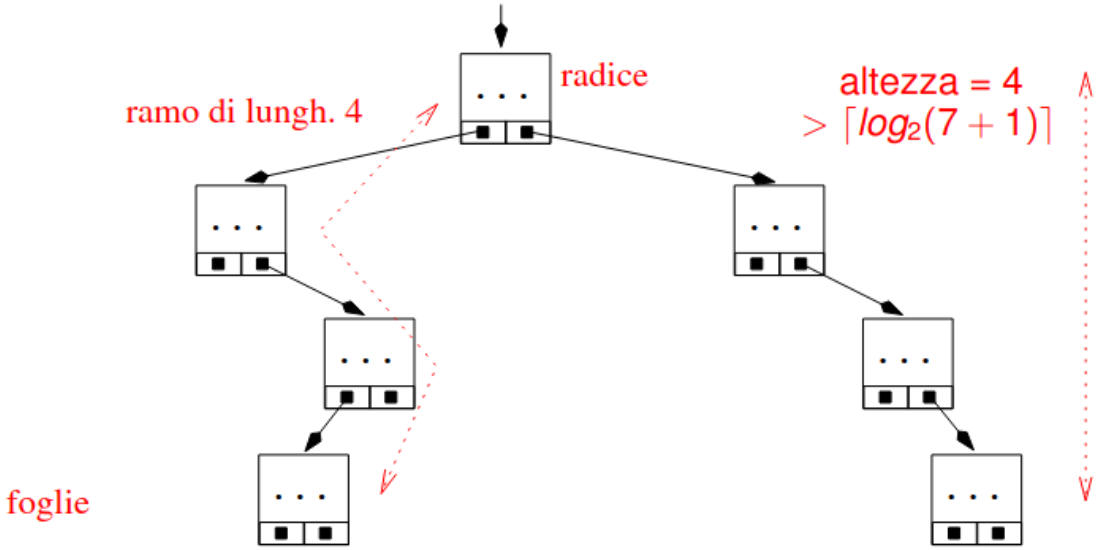
Un albero binario di N elementi è ***bilanciato*** se la sua altezza è $h = log2(N + 1)$: tutti i rami hanno lunghezza *h* o *h-1*.

Un albero binario di *N* elementi è ***completo*** se la sua altezza è tale che $N = 2^{(h)} - 1$: tutti i rami hanno lunghezza *h*.

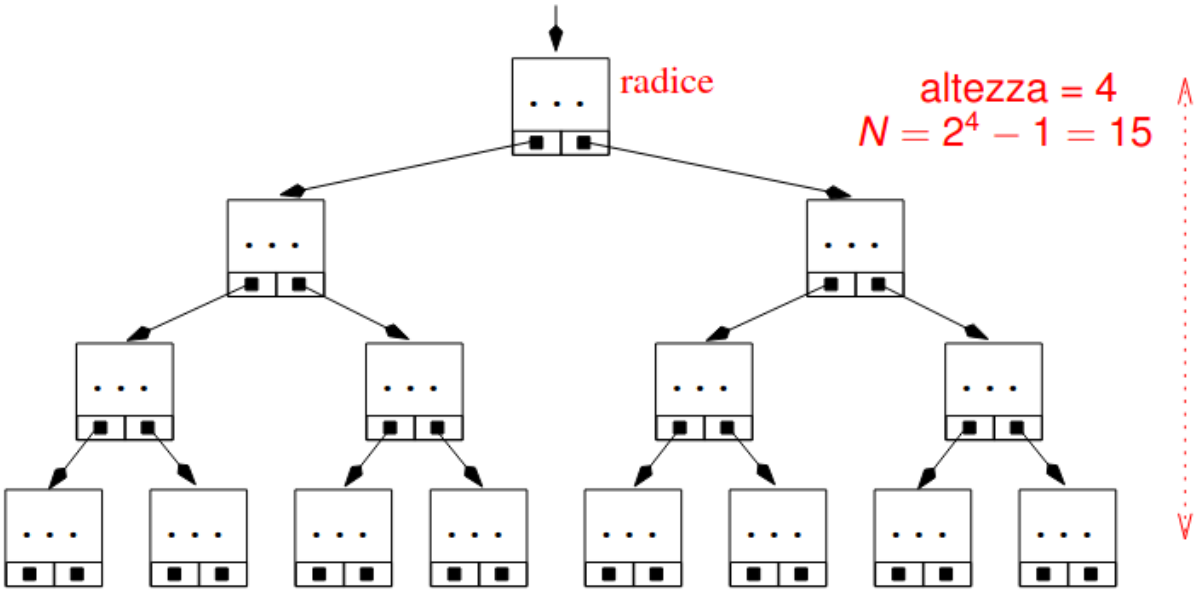
57.3.1.1 Esempio di albero binario bilanciato



57.3.1.2 Esempio di albero non bilanciato



57.3.1.3 Esempio di albero completo



57.3.2 Albero di ricerca binaria

Un albero di ricerca binaria è una struttura dati utile a mantenere dati ordinati.
Assumiamo una relazione di ordine totale di precedenza "<=" tra gli elementi *T,* ad esempio: ordine numerico, ordine alfabetico del campo "cognome", ecc.

Definizione

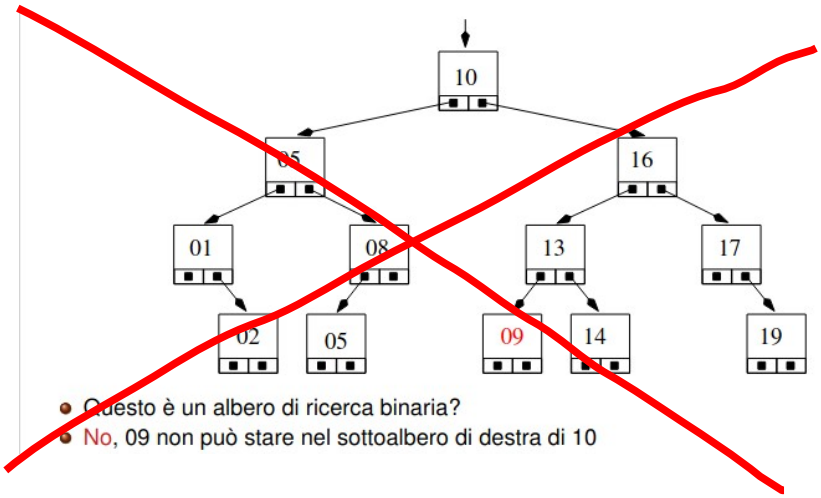
Un albero binario è un albero di ricerca binaria se ogni nodo N dell'albero verifica la seguente proprietà:

- Tutti i nodi del sottoalbero di sinistra precedono strettamente *N*
- Tutti i nodi del sottoalbero di destra sono preceduti da *N*

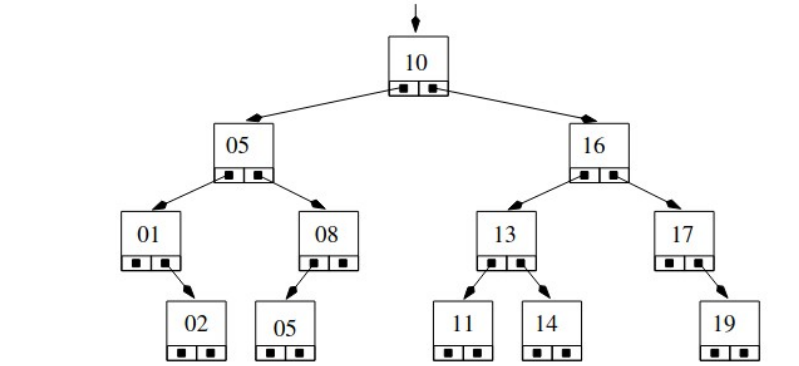
(è possibile invertire lo "strettamente" tra sinistra e destra).

Nota: in alcuni casi non è previsto che ci possano essere due valori uguali nel valore valutato dalla relazione di precedenza (valore chiave).

57.3.2.1 Esempio: albero di ricerca binaria

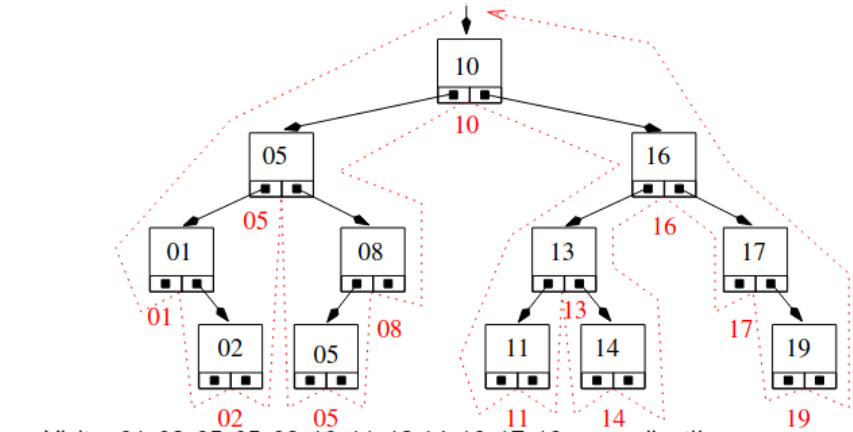


- Questo è un albero di ricerca binaria?
- No, 09 non può stare nel sottoalbero di destra di 10



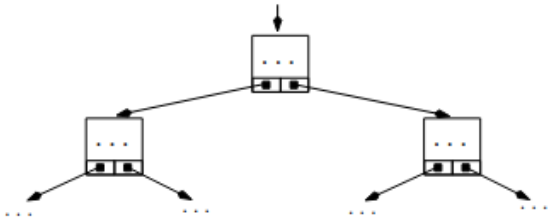
- Questo è un albero di ricerca binaria?
- Sì

57.3.2.2 Esempio: visita ordinata di un albero di ricerca binaria.



- Visita: 01, 02, 05, 05, 08, 10, 11, 13.14, 16, 17, 19 $\Longrightarrow$ ordinati!

57.4 Implementazione di un albero di ricerca binaria.



Dati: un albero di ricerca binaria *t*

- *t* punta al primo elemento inserito nell'albero (inizialmente NULL)
  - albero vuoto: *t==NULL*
  - albero pieno: out of memory (numero di elementi contenuti nell'albero limitato solo dalla memoria)

N.B.: allocati solo gli n nodi necessari a contenere gli elementi**!**

Funzionalità:
init: pone *t=NULL*

search (cerca un elemento *val* in *t*):

1. se *t == NULL*, restituisce *NULL*
2. se *val == t->value*, restituisce *t*
3. se *val < t->value*, cerca ricorsivamente in *t->left*
4. se *val > t->value*, cerca ricorsivamente in *t->right*

insert (inserisce un elemento *val* in *t*):

1. se *t* è vuoto, *t == NULL*:
   - crea un nuovo nodo per il puntatore *tmp*
   - pone *tmp->value=val*, *tmp->left=NULL*, *tmp->right=NULL*,
   - pone *t=tmp*
2. se *val < t->value*, inserisci ricorsivamente in *t→left*
3. se *val >=t->value*, inserisci ricorsivamente in *t→right*

print (stampa in modo ordinato l'albero *t*):
Se l'albero non è vuoto:

- stampa ricorsivamente il sottoalbero sinistro *t->left*
- stampa il contenuto del nodo puntato da *t*: *t->value*
- stampa ricorsivamente il sottoalbero destro *t->right*

deinit: s
Se l'albero non è vuoto:

- applica ricorsivemente *deinit* ai sottoalberi sinistro *t->left* e destro *t->right*
- applica *delete* al nodo puntato da *t*

remove: per rimuovere un elemento. Da analizzare.

57.5 Implementazione in C++

File ***.h***: dichiarazioni di funzioni e della struttura.
enum retval {FAIL,OK};

Struttura dei nodi
struct node;
*typedef node * tree;*
struct node
{
char item; //è un albero di *char*
tree left;
tree right;
};

Dichiarazione delle funzioni

void init(tree &);
void deinit(tree &);
bool nullp(const tree &);
retval insert(tree &, char);
tree cerca (const tree &,char);
void print_ordered(const tree &);
void print_indented(const tree &); //stampa l'albero in modo simpatico, con le indentazioni

File ***.cc***: implementazione delle funzioni.

Funzioni ausiliarie
```
static void print_spaces(int depth) {
  for(int i=0;i<depth;i++)
      cout << "  ";
}

static bool emptyp(const tree & t) {          //verifica se è vuota
  return (t==NULL);
}
```

Funzioni imperative
init, deinit
```
void init(tree & t) {
  t=NULL;
}

void deinit(tree & t) {
  if (!emptyp(t)) {
    deinit(t->left);
    deinit(t->right);
    delete t;
  }
}
```
insert (per inserire gli elementi fare n volte insert dove n sono gli elementi da inserire)
retval insert(tree & t, char v) {

```
  retval res;
  if (emptyp(t)) {
// memo: "new (nothrow) ..." restituisce NULL
// senza generare eccezioni se l'allocazione non va a buon fine
    t = new (nothrow) node;
    if (t==NULL)
      res = FAIL;
    else {
      t->item = v;
      t->left = NULL;
      t->right = NULL;
      res = OK;
    }
  }
  else if (v <= t->item)
    res = insert(t->left, v);
  else if (v > t->item)
    res = insert(t->right, v);
  return res;
}
```

cerca, nullp

```
tree  cerca (const tree & t,char elem) {
  tree res;
  if (emptyp(t))
    res = NULL;
  else if (elem==t->item)
    res = t;
  else if (elem < t->item)
    res = cerca(t->left,elem);
  else if (elem > t->item)
    res = cerca(t->right,elem);
  return res;
}
```

//restituisce un puntatore al nodo trovato o NULL se non lo trova

// valuta il valore ritornato da '*cerca*' → ritorna *true* se non è stato trovato l'elemento

```
bool nullp(const tree & t) {
  return (t==NULL);
}
```

print_orderdered, print_indented

```
void print_ordered(const tree & t) {
  if (!emptyp(t)) {
    print_ordered(t->left);
    cout << t->item << endl;
    print_ordered(t->right);
  }
}

void print_indented(const tree & t) {
  static int depth=0;
  depth++;
```

```
  if (!emptyp(t)) {
    print_indented(t->right);
    print_spaces(depth);
    cout << t->item << endl;
    print_indented(t->left);
  }
  depth--;
}
```

Nel file ***main***: includere la libreria e definire una variabile ***tree t***. Ricordarsi *init* e *deinit*.

57.6 Implentazione di un albero binario tramite array

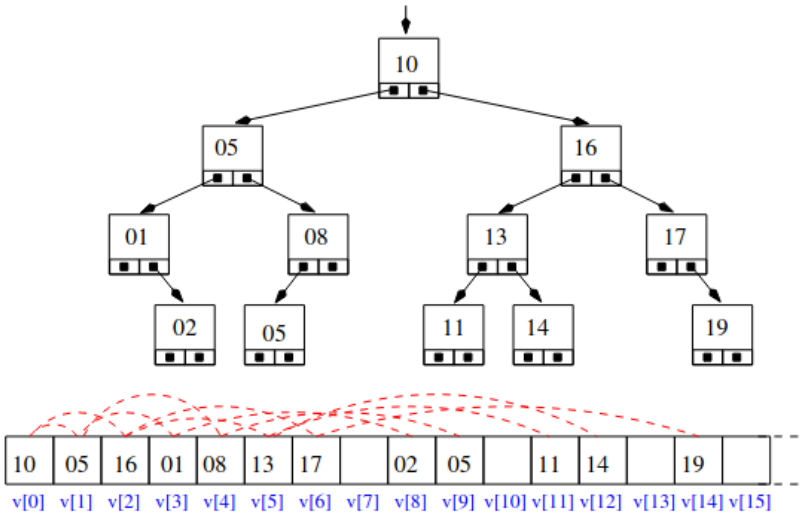
Dati: un array *v* di *dim* elementi di tipo *T*.
Un (sotto)albero è dato da un puntatore a *v* e un indice *i*
struct tree { T * v; int i; };

- *v* allocato dinamicamente
- l'elemento radice è in *v[0]*
- se un elemento è in posizione *v[i]*, i suoi due figli sono in posizione *v[2*i+1]* e *v[2*i+2]*

Necessaria una nozione ausiliaria di "elemento vuoto".

Funzionalità: come nell'implementazione precedente, cambia solo la nozione di figlio sinistro/destro.

N.B.! Vengono allocati *dim* nodi: efficace solo se ben bilanciato!



Nel file ***.h***: definizione struct, funzioni e MAX DIM.
Enum e max dim.

```
enum retval {FAIL,OK};
const int MAXSIZE = 100;

// "puntatore"
struct tree {
  char * array;
  int pos;
};
```

Funzioni

```
void init(tree &);
void deinit(tree &);
bool nullp(const tree & );
retval insert(tree &, char);
tree cerca (const tree &,char);
void print_ordered(const tree &);
void print_indented(const tree &);
```

Nel file ***.cc***: implementazione di tutte le funzioni.

FUNZIONI AUSILIARIE

right,left (determina il nodo destro e sinistro)

```
static tree left(const tree & t) {
  tree res;
  res.pos = 2*t.pos+1;
  res.array=t.array;
  return res;
}

static tree right(const tree & t) {
  tree res;
  res.pos = 2*t.pos +2;
  res.array=t.array;
  return res;
}
```

elem (ritorna l'elemento in quella posizione)

```
static char & elem(const tree & t) {
  return t.array[t.pos];
}
```

emptyp (verifica se l'albero è vuoto)

```
static int emptyp(const tree & t) {
  return (elem(t)=='\0');
}
```

nullp, mknull,print_spaces

```
bool nullp(const tree & t) {
  return (t.pos==-1);
}
```

//usato nella funzione <u>cerca</u>, se l'albero è vuoto

```
static tree mknull() {
  tree res;
  res.pos=-1;
  return res;
}

static void print_spaces(int depth) {
  for(int i=0;i<depth;i++)
     cout << "  ";
}
```

FUNZIONI IMPERATIVE

init e deinit

```
void init(tree & t) {
  t.pos=0;
  t.array = new char[MAXSIZE];
  for (int i=0;i<MAXSIZE;i++)
    t.array[i] = '\0';
}

void deinit(tree & t) {
  delete [] t.array;
}
```

insert

```
retval insert(tree & t, char v) {
  tree t1;
  retval res;
  if (t.pos >= MAXSIZE)
    res = FAIL;
  else if (emptyp(t)) {
    elem(t)=v;
    res = OK;
  }
  else if (v <= elem(t)) {
    t1 = left(t);
    res = insert(t1,v);
  }
  else {
    t1 = right(t);
    res = insert(t1,v);
  }
  return res;
}
```

print

```
void print_ordered(const tree & t) {
  if (!emptyp(t)) {
    print_ordered(left(t));
    cout << elem(t) << endl;
    print_ordered(right(t));
  }
}

void print_indented(const tree & t) {
  static int depth=0;
  depth++;
  if (!emptyp(t)) {
    print_indented(right(t));
    print_spaces(depth);
    cout << elem(t) << endl;
    print_indented(left(t));
  }
```

```
  depth--;
}
```

cerca

```
tree cerca (const tree & t,char item) {
  tree res;
  if (emptyp(t)) {
    res = mknull();
  }
  else if (elem(t)==item) {
    res = t;
  }
  else if (elem(t)>item) {
    res = cerca(left(t),item);
  }
  else {
    res = cerca(right(t),item);
  }
  return res;
}
```

Nel ***main*** creare una variabile ***tree t***.

57.7 Albero binario modulare: per creare alberi di ogni tipo si voglia
Le varie funzioni che regolano tutte le funzionalità dell'albero rimangono invariate. Cambiano solo ***insert*** e ***cerca*** e ***print_ordered*** e ***print_indented***, che vengono adattate alla modularità affinché, cambiano solo un parametro in un file header, si riesca a definire un albero solo di *char*, solo di *int*, solo di *float* e via dicendo.

Il trucco che sta alla base di tutto ciò consiste nel sostituire alle funzioni il tipo *char*, *int* ecc. con il tipo fittizio ***contenuto*** (un alias di *char*, se abbiamo un albero di *char*, di *int*, con un albero di *int*, e così via). Ci servirà un altra coppia di file che definisca la modularità.

FILE PER LA MODULARITÀ
module.h (da includere in ***tree.cc***, ove sono definite le funzioni per l'albero).

Il segreto per la modularità
typedef char contenuto;
typedef char chiave;
//cambiando *char* in *int* avremo un albero di interi

Funzioni per la modularità
chiave chiaveDi(contenuto);
int confronta(chiave, contenuto) ;
void stampa(const contenuto&) ;

module.cc (con la definizione delle funzioni per la modularità)
```
chiave chiaveDi(contenuto v) {
  return (chiave) v;
}

// restituisce rispettivamente -1,0,1
// se c e' piu' piccola, uguale o piu' grande della chiave di v1
int confronta(chiave c,contenuto v1) {
  int res;
  if (c<v1)
    res = -1;
  else if (c>v1)
    res = 1;
  else // (c==v1)
    res = 0;
  return res;
}

void stampa(const contenuto & v) {
  cout << v << endl;
}
```

VARIAZIONI IN *tree.cc*

insert e *cerca*

```
retval insert(tree & t, contenuto v) {          //come parametro si usa contenuto
  retval res;
  if (emptyp(t)) {
```

```
// memo: "new (nothrow) ..." restituisce NULL
// senza generare eccezioni se l'allocazione non va a buon fine
    t = new (nothrow) node;
    if (t==NULL)
      res = FAIL;
    else {
      t->item = v;
      t->left = NULL;
      t->right = NULL;
      res = OK;
    }
  }
  else if (confronta(chiaveDi(v),t->item)<=0)
    res = insert(t->left, v);
  else if (confronta(chiaveDi(v),t->item) > 0)
    res = insert(t->right, v);
  return res;
}

tree  cerca (const tree & t,chiave c) {
  tree res;
  if (emptyp(t))
    res = NULL;
  else if (confronta(c,t->item)==0)          //per confrontare gli elementi
    res = t;
  else if (confronta(c,t->item)<0)
    res = cerca(t->left,c);
  else if (confronta(c,t->item)>0)
    res = cerca(t->right,c);
  return res;
}
```

print_ordered e *print_indeted* //in queste due non si usa più un *cout*, ma la funzione *stampa(contenuto)* di *module.h*

```
void print_ordered(const tree & t) {
  if (!emptyp(t)) {
    print_ordered(t->left);
    stampa(t->item);
    print_ordered(t->right);
  }
}

void print_indented(const tree & t) {
  static int depth=0;
  depth++;
  if (!emptyp(t)) {
    print_indented(t->right);
    print_spaces(depth);
    //stampa(t->item);
    cout << t->item << endl;
    print_indented(t->left);
  }
  depth--;}
```

Nel file ***tree.h*** ovviamente sarà necessario cambiare la dichiarazione di *insert* e *cerca*, oltre che il tipo del campo *item* della struttura che crea l'albero (sarà di tipo ***contenuto***).

Nel file ***main*** è importante che la variabile che contiene l'elemento che l'utente voglia inserire sia anch'essa di tipo ***contenuto*** e non di un tipo definito.

In questo modo cambiando i due ***typedef*** nel file *module.h* si riesce a cambiare semplicemente il tipo di albero binario che si va a creare