Alberi Binari di Ricerca di Strutture

- Obiettivi:
  - Riprendere la nozione di strutture dato elementari per la gestione di collezioni di oggetti
  - Strutture dato tavole (tabelle, liste), ma realizzate con alberi binari di ricerca

Strutture dati

- Termine usato per indicare insiemi (collezioni) con elementi del dominio compositi, ad esempio:
  - tavole (dizionari);
  - liste;
  - insiemi;
  - pile e code;
  - Alberi binari di ricerca e grafi
- corredate delle operazioni per la loro gestione (efficiente)

Il tipo dato Dizionario (Tavola)

tipo `Dizionario`:
dati:
un insieme S di coppie $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k).

`delete`$(chiave\ k)$
cancella da S la coppia con chiave k.

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce l'elemento e ad essa associato, e `null` altrimenti.

Dizionario (tavole)

- Tipo di dato astratto per rappresentare insiemi di coppie:

  <chiave, attributi>

- Realizzabile – in memoria centrale – con alberi binari di ricerca (strutture dati collegate)

Nome	Cognome	Reddito	AliquotaMax
Mario	Rossi	18324	27
Luca	Bianchi	1453253	43

Esercitazione 6.1 *rivisitata*

- Scrivere un programma che realizzi una rubrica telefonica come albero binario di ricerca in memoria centrale. In particolare, ogni elemento della rubrica è caratterizzato dalle seguenti informazioni:
  - Nome (chiave)
  - numero_telefono
- Il programma deve essere in grado di attuare varie richieste dell'utente:
- ***inserimento***: l'utente vuole inserire un nuovo record nell'archivio. Dati nome e numero di telefono della persona da inserire, il programma aggiunge un nuovo elemento all'albero binario di ricerca (ovvero all'archivio).

Esercitazione 6.1 (*cont.*)

- ***cancellazione***: l'utente vuole eliminare un elemento dall'archivio. Dato un nome, il programma dovrà eliminare (se esiste) l'elemento con il nome specificato dall'albero.
- ***ricerca***: dato in ingresso il nome di una persona presente in archivio, si richiede la visualizzazione del numero di telefono relativo alla persona data;
- ***uscita***: l'utente richiede che il programma termini.

- L'interazione tra l'utente e il programma avviene in modo ciclico: l'utente può sottoporre una richiesta ad ogni ciclo ed il programma, sfruttando un meccanismo di selezione (per esempio **switch**) reagisce nel modo richiesto. L'esecuzione del programma si conclude quando l'utente richiede l'uscita.

Soluzione (come BinarySearchTree)

Rappresentazione collegata, come un albero binario di ricerca di strutture (in memoria centrale). Il tipo di ciascun elemento della rubrica/albero binario di ricerca è sempre lo stesso già visto:

```
typedef struct {  char  nome[20];
                  char  tel[16];    } elemento;
```

Il BST che rappresenta la rubrica è quindi del tipo:

```
typedef struct list_element {
                elemento value;
                struct list_element *left;
                struct list_element *right; } item;
typedef item *tree;
```

Una variabile di tipo tree mantiene il valore del puntatore radice; la ***dimensione logica*** è data dal numero di nodi dell'albero:

```
tree R;
```

Dichiarazioni globali

```
#include <stdio.h>
#include <string.h>
typedef struct {  char  nome[20];
                  char  tel[16];    } elemento;

typedef struct list_element {
              elemento value;
              struct list_element *left;
              struct list_element *right; } item;
typedef item *tree;

/* prototipi */
int menu(void);
tree inserimento(tree R);
tree cancellazione(tree R);
void ricerca(tree R);
```

main

```
void main(void)
{int scelta, fine;
 tree R = NULL;                    /* R è la radice */

 fine=0;
 do
 { scelta=menu();
   switch(scelta){
     case 1: R = inserimento(R); break;
     case 2: R = cancellazione(R); break;
     case 3: ricerca(R);  break;
     case 4: fine=1; break;
     default: printf("Scelta sbagliata\n");
              }
      }while (!fine);
}
```

menu

```
int menu(void)
{     int ris;
      printf("Scegli l'operazione:\n");
      printf("\t1\tInserimento\n");
      printf("\t2\tCancellazione\n");
      printf("\t3\tRicerca\n");
      printf("\t4\tUscita\n");
      printf("\n\nScelta:  ");
      scanf("%d", &ris);
      return ris;
}
```

Inserimento (ordinato in BST)

L'inserimento determina la memorizzazione dell'elemento dato da standard input nella posizione deputata nel BST:

```
tree inserimento (tree R)
{     elemento el;
      printf("\nInserire nome:  ");
      scanf("%s",el.nome);
      printf("\nInserire numero:  ");
      scanf("%s",el.tel);
      return ordins_it(el,R);
}
```

Non c'è il problema di riempire tutte le locazioni ... Finché c'è memoria heap disponibile si inserisce (come con la soluzione con lista)

Inserimento in BST (*iterativo*)

```
tree ordins_it(elemento e,tree root)
{tree p=NULL, t=root;       // p predecessore o nodo padre
 if (root==NULL) return cons_tree(e,NULL,NULL);
 else
    { while (t!=NULL)
        if (strcmp(el.nome,t->value.nome)<=0)
          {p=t;  t=t->left;}
        else
          {p=t;  t=t->right;}
    }                                  //p punta a un nodo foglia
      if (strcmp(el.nome,p->value.nome)<=0)
          p->left = cons_tree(e,NULL,NULL);
      else
          p->right = cons_tree(e,NULL,NULL);
 return root;  }
```

Inserimento in BST (*ricorsivo*)

```
tree ord_ins(elemento e, tree t)
{                    //BST con duplicazioni
if (t==NULL)
      return(cons_tree(e,NULL,NULL) );
  else
    { if (strcmp(el.nome,t->value.nome)<=0)
              t->left = ord_ins(e,t->left);
          else
               t->right = ord_ins(e,t->right);
    return t;
    }
}
```

In caso di chiave composta …

Se la chiave fosse costituita da più attributi, ad esempio **nome** e **cognome** con ordinamento sul campo **cognome** e a parità di **cognome**, sul campo **nome**:

```
tree ord_ins(elemento e, tree t)
{ if (t==NULL)
      return(cons_tree(e,NULL,NULL) );
  else
   { if ( (strcmp(el.cognome,t->value.cognome)<0) ||
          ( !strcmp(el.cognome,t->value.cognome) &&
            (strcmp(el.nome,t->value.nome)<=0)  )  )
           t->left = ord_ins(e,t->left);
        else
           t->right = ord_ins(e,t->right);
    return t;
    }
}
```

ricerca

Ricerca nella rubrica (data come parametro) del telefono di un elemento attraverso il suo nome (letto da input):

```
void ricerca(tree R) {
  tree k=NULL;          /* qui k puntatore di tipo tree */
  elemento e;
  printf("\nInserire nome:  ");
  scanf("%s",e.nome);
  k=individua(R,e);      /* restituisce un puntatore */
  if (k!=NULL)
   {  printf("\nTelefono di %s ...\n", k->value.nome);
      printf("\è  %s ...\n", k->value.tel);
            }
  else printf("\n%s\t non trovato\n", e.nome);
}
```

Individua (simile a *member_ord*)

Simile a member_ord_it, realizza la ricerca binaria (in versione iterativa) e restituisce il puntatore al nodo se trovato, NULL altrimenti:

```
tree individua(elemento el, tree t)
{ while(t!=NULL)
    { if (!strcmp(el.nome,t->value.nome)) return t;
      else
         if (strcmp(el.nome,t->value.nome)<0)
                          t=t->left;
         else
                          t=t->right;
    }
  return NULL;
}
```

In caso di chiave composta ...

Se la chiave fosse costituita da più attributi, ad esempio **nome** e **cognome** con ordinamento sul campo **cognome** e a parità di **cognome**, sul campo **nome**:

```
tree individua(elemento el, tree t)
{ while(t!=NULL)
  { if ((!strcmp(el.cognome,t->value.cognome) &&
        (!strcmp(el.nome,t->value.nome) ) return t;
    else
       if ( (strcmp(el.cognome,t->value.cognome)<0) ||
           ( !strcmp(el.cognome,t->value.cognome) &&
              (strcmp(el.nome,t->value.nome)<=0)  )  )
                  t=t->left;
         else     t=t->right;
    }
  return NULL;  }
```

cancellazione

Eliminazione dalla rubrica (data come parametro) dell'elemento identificato attraverso il suo nome (letto da input):

```
tree cancellazione(tree R)
{ elemento e;
  printf("\nInserire nome:  ");
  scanf("%s",e.nome);
  return delete(el,R);
}
```

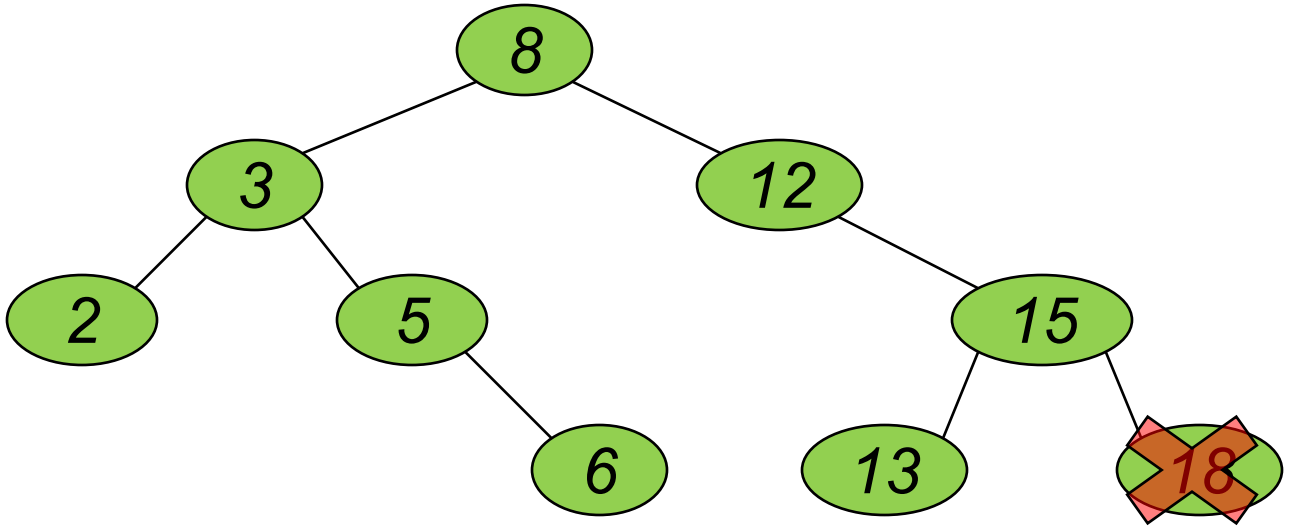
La cancellazione di un nodo non è così facile come l'inserimento (non la vediamo ...)

Può essere realizzata in modo iterativo, considerando tre casi

Algoritmo - cancellazione

Caso 1: il nodo da rimuovere è una foglia (18)

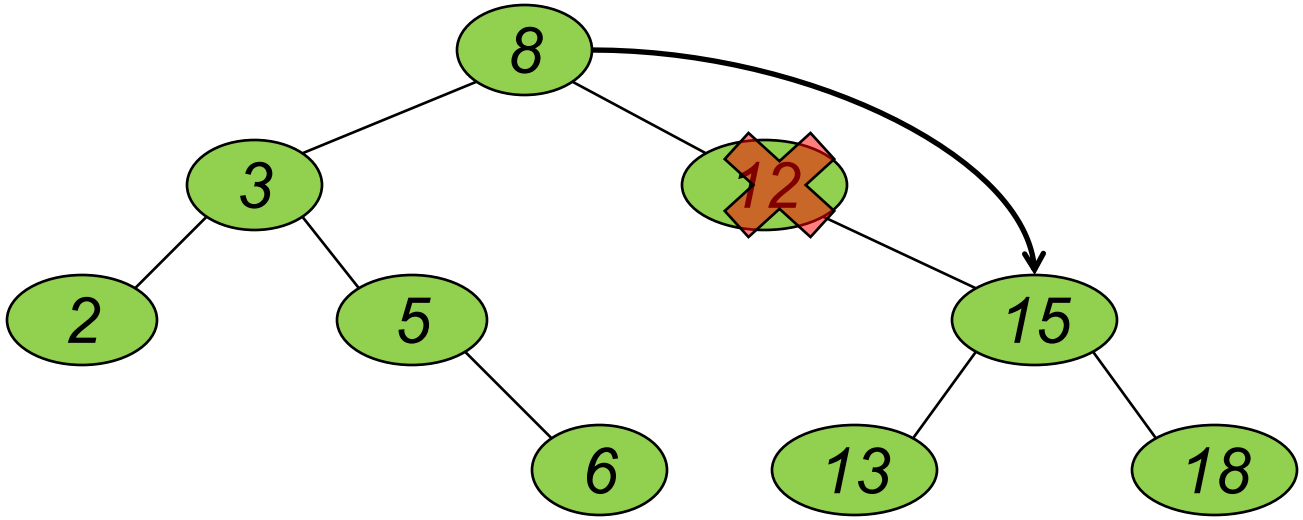
- Ricerca del nodo (salvando sempre il puntatore padre)
- Rimozione dal padre



Algoritmo - cancellazione

Caso 2: il nodo da rimuovere ha un solo figlio

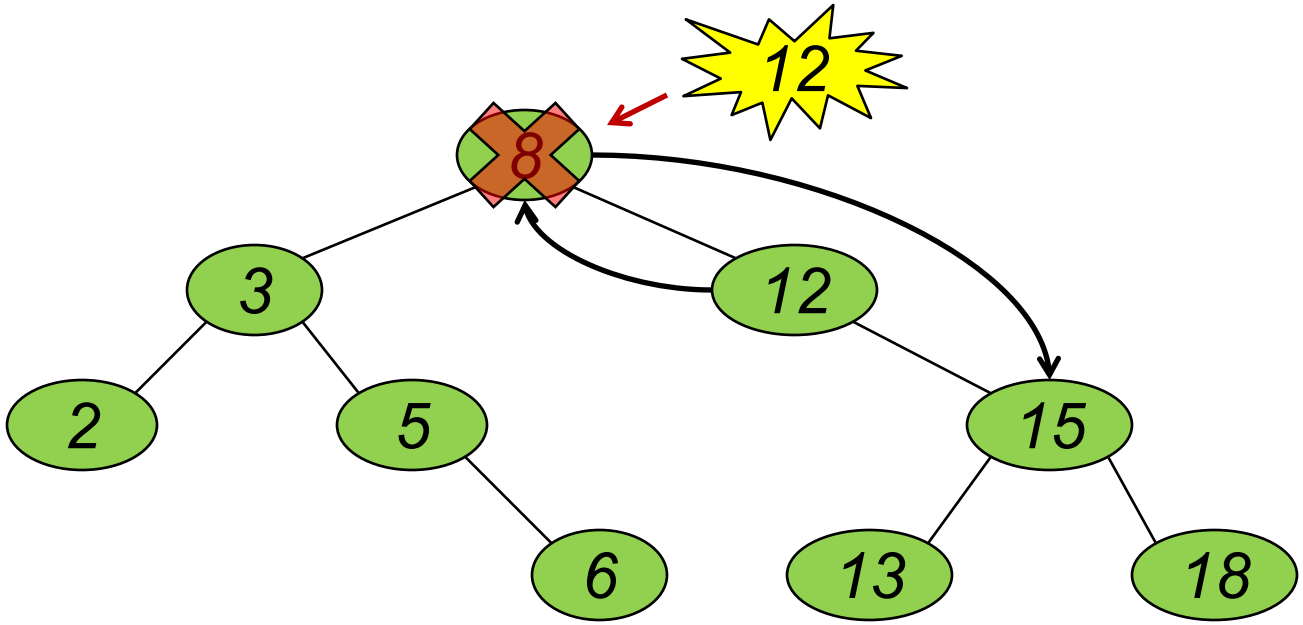
- Ricerca del nodo
- Rimozione dal padre e il padre punta ora all'unico figlio del nodo rimosso



Algoritmo - cancellazione

Caso 3: il nodo da rimuovere (8) ha due figli

- Ricerca del nodo
- Sostituzione del valore del nodo da rimuovere con il valore del più piccolo fra i valori maggiori (il nodo più a sinistra del sotto-albero di destra)



Cancellazione in BST

per chi vuole approfondire, si rimanda al sito:

`https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/`

Dizionario (tavola) come BST

- Vantaggi:
  - L'albero binario di ricerca è una struttura dati dinamica, cresce finché c'è spazio nell'heap (come era per la realizzazione tramite lista)
  - Cambia il tipo degli elementi (campo value), ma le operazioni in pratica restano quasi le stesse (tranne i confronti, *sul campo chiave)*
  - Possiamo applicare la ricerca binaria e, se BST bilanciato, con complessità $O(\log_2 N)$ con N numero dei nodi dell'albero