Strutture Dati

Lezione 13
Alberi binari di ricerca

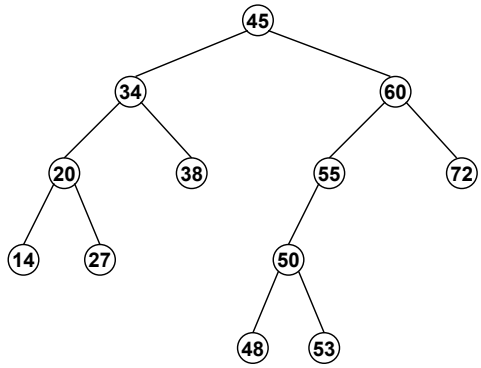
Oggi parleremo di …

- Albero binario di ricerca
  - definizione
  - rappresentazione
  - operazioni
    - ricerca
    - inserimento
    - cancellazione

Albero binario di ricerca: definizione

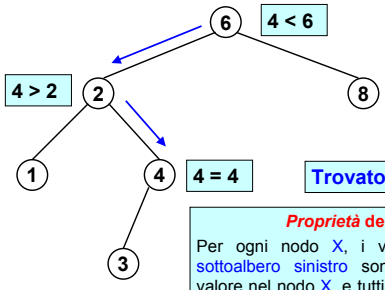
- E' utilizzato per la gestione e ricerche in grosse quantità di dati
  - liste ed array non sono adeguati perché inefficienti in tempo O(*n*) o in spazio;
  - mantenimento di archivi (*DataBase*);
  - mantenimento e gestione di corpi di dati su cui si effettuano molte ricerche.
- Un **albero binario di ricerca** è un albero in cui
  - le chiavi sono uniche;
  - le chiavi del sottoalbero sinistro non vuoto sono più piccole della chiave della radice;
  - le chiavi del sottoalbero destro non vuoto sono più grandi della chiave della radice;
  - i sottoalberi destro e sinistro sono alberi binari di ricerca.

Albero binario di ricerca: esempio



Albero binario di ricerca: esempio

Ricerca del valore 4



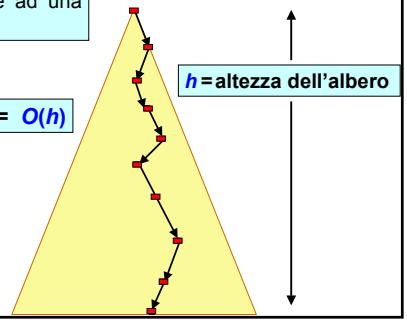
Proprietà* degli *ABR

Per ogni nodo X, i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X, e tutti i valori nei nodi del sottoalbero destro sono maggiori del valore di X.

Albero binario di ricerca: ricerca

La ricerca è confinata ai nodi posizionati lungo un singolo percorso dalla radice ad una foglia.

Tempo di ricerca = *O*(*h*)



Albero binario di ricerca: rappresentazione

- È una specializzazione dell'ADT albero binario.
- La specifica è la stessa, l'unica differenza è che si assume che i dati contenuti (le chiavi) siano ordinabili secondo qualche relazione d'ordine.
- E' rappresentato mediante puntatori.


```
typedef char elemento;
typedef struct nodo *tree_pointer;
typedef struct nodo {
        elemento key;
        tree_pointer figlio_sinistro;
        tree_pointer figlio_destro;
        } albero;

tree_pointer radice = NULL;
```

Albero binario di ricerca: ricerca

- Se la radice è NULL, la ricerca non ha successo.
- Altrimenti, si confronta la chiave da ricercare con la chiave della radice
  - se i due valori sono uguali, la ricerca termina con successo;
  - se la chiave è minore della chiave della radice, allora si effettua la ricerca nel sottoalbero sinistro della radice;
  - se la chiave è maggiore della chiave della radice, allora si effettua la ricerca nel sottoalbero destro della radice.

Albero binario di ricerca: ricerca

Versione ricorsiva

```
tree_pointer  ric_ric(tree_pointer radice, int chiave)
{
   /* fornisce un puntatore al nodo che contiene item
      se tale nodo non esiste fornisce NULL */

   if (!radice) return(NULL);
   if (chiave == radice->dati)  return(radice);
   if (chiave < radice->dati)   return(ric_ric(radice->figlio_sinistro, chiave));
   return(ric_ric(radice->figlio_destro, chiave));
}
```

Versione iterativa

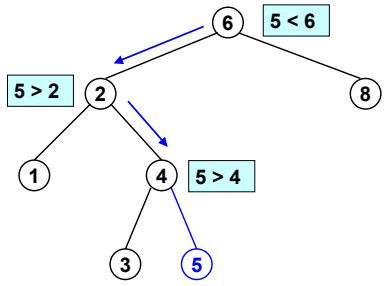
```
tree_pointer  ric_iter(tree_pointer tree, int chiave)
{
   /* fornisce un puntatore al nodo che contiene item
      se tale nodo non esiste fornisce NULL */

   while(tree) {
     if (chiave == tree->dati)  return(tree);
     if (chiave < tree->dati)   tree = tree->figlio_sinistro;
     else  tree = tree->figlio_destro;
    }
   return(NULL);
}
```

$O(h)$

Albero binario di ricerca: inserimento

Inserimento del valore 5



Albero binario di ricerca: inserimento

- Si effettua una ricerca nell'albero.
- Se la ricerca non ha successo, si inserisce l'elemento nel punto in cui è terminata la ricerca.
- Si utilizza una versione modificata della funzione ricerca
  - restituisce NULL se l'albero è vuoto o l'elemento è già presente;
  - restituisce il puntatore all'ultimo nodo dell'albero incontrato durante la ricerca.
- Il nuovo elemento viene inserito come figlio di tale nodo.

Albero binario di ricerca: inserimento

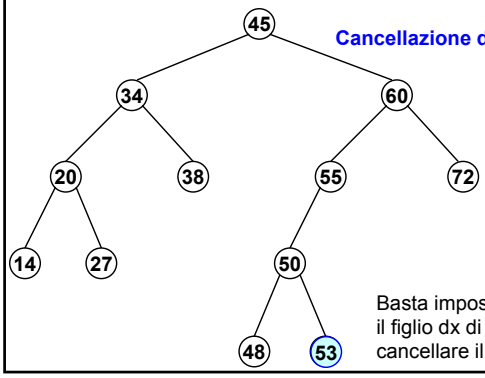
```
void insert_nodo(tree_pointer *nodo, int num)
{
  tree_pointer ptr, temp;

  temp = ric_modificata(*nodo, num);
  if(temp || !(*nodo)) {
        ptr = (tree_pointer)malloc(sizeof(nodo));
        if (ptr == NULL) {
                printf("\la memoria è piena");
                exit(1);
                }
        ptr->dati = num;
        ptr->figlio_sinistro = NULL;
        ptr->figlio_destro = NULL;
        if(*nodo)
                if(num < temp->dati)      temp->figlio_sinistro = ptr;
                else  temp->figlio_destro = ptr;
        else    *nodo = ptr;
      }
}
```

$O(h)$

Albero binario di ricerca: cancellazione

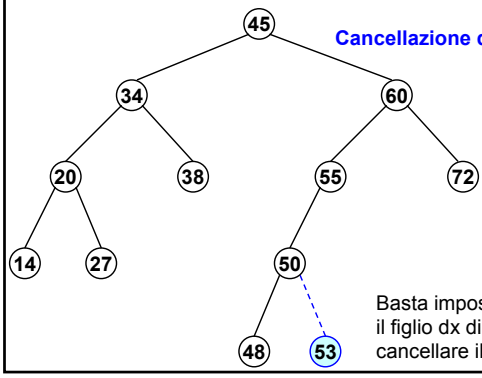
Caso 1: cancellazione di un nodo terminale



Basta impostare a NULL il figlio dx di 50 e cancellare il nodo!

Albero binario di ricerca: cancellazione

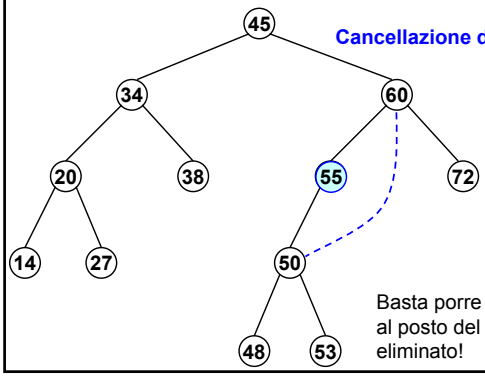
Caso 1: cancellazione di un nodo terminale



Basta impostare a NULL il figlio dx di 50 e cancellare il nodo!

Albero binario di ricerca: cancellazione

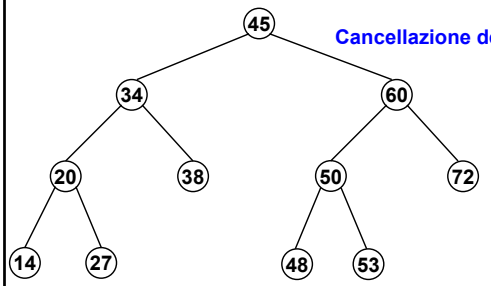
Caso 2: cancellazione di un nodo di grado 1



Basta porre l'unico figlio al posto del nodo eliminato!

Albero binario di ricerca: cancellazione

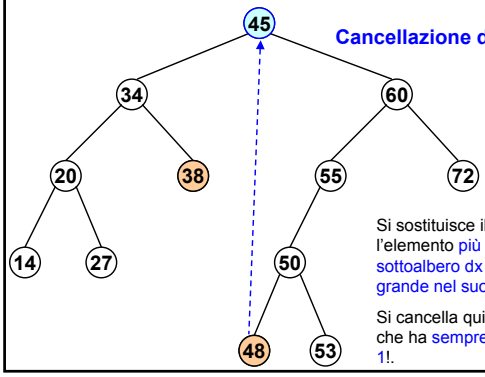
Caso 2: cancellazione di un nodo di grado 1



Basta porre l'unico figlio al posto del nodo eliminato!

Albero binario di ricerca: cancellazione

Caso 3: cancellazione di un nodo con grado 2

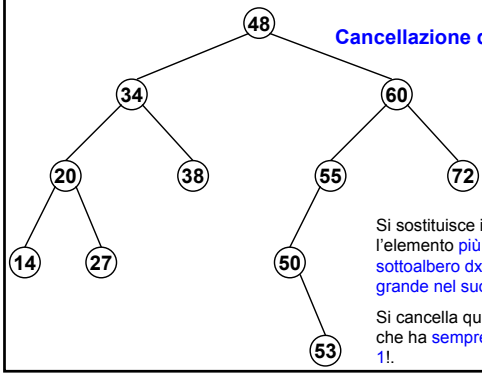


Si sostituisce il nodo con l'elemento più piccolo nel suo sottoalbero dx o con il più grande nel suo sottoalbero sx.

Si cancella quindi tale nodo che ha sempre grado 0 oppure 1!.

Albero binario di ricerca: cancellazione

Caso 3: cancellazione di un nodo con grado 2

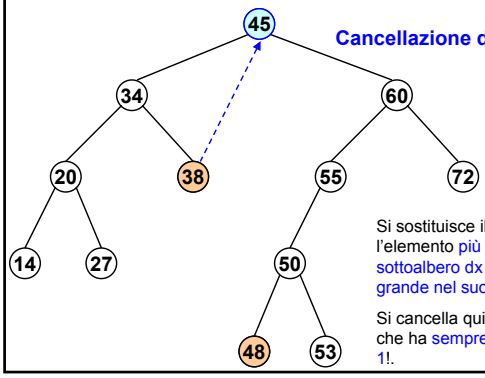


Si sostituisce il nodo con l'elemento più piccolo nel suo sottoalbero dx o con il più grande nel suo sottoalbero sx.

Si cancella quindi tale nodo che ha sempre grado 0 oppure 1!.

Albero binario di ricerca: cancellazione

Caso 3: cancellazione di un nodo con grado 2



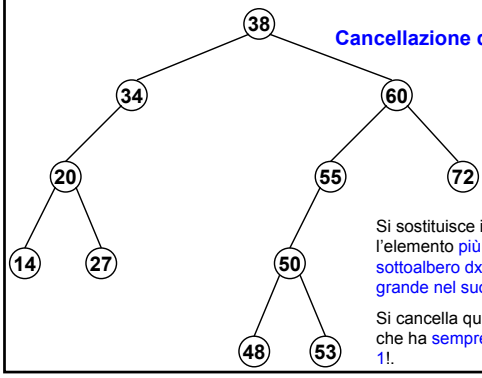
Cancellazione del valore 45

Si sostituisce il nodo con l'elemento più piccolo nel suo sottoalbero dx o con il più grande nel suo sottoalbero sx.

Si cancella quindi tale nodo che ha sempre grado 0 oppure 1!.

Albero binario di ricerca: cancellazione

Caso 3: cancellazione di un nodo con grado 2



Cancellazione del valore 45

Si sostituisce il nodo con l'elemento più piccolo nel suo sottoalbero dx o con il più grande nel suo sottoalbero sx.

Si cancella quindi tale nodo che ha sempre grado 0 oppure 1!.

Albero binario di ricerca: cancellazione

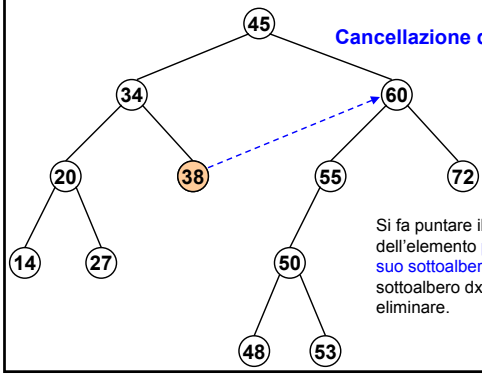
Per copiatura

```
void delete_nodo_copia(tree_pointer *radice, int item)
{
 tree_pointer tmp, node, previous, prev=NULL, ptr=*radice;
 while (ptr!=NULL && item!=ptr->dati)
 {
   prev = ptr;
   if(item>ptr->dati)         ptr = ptr->figlio_destro;
   else    ptr = ptr->figlio_sinistro;
 }
 node = ptr;
 if (ptr!=NULL &&  item==ptr->dati) {
      if (node->figlio_destro==NULL)    node = node->figlio_sinistro;
      else if (node->figlio_sinistro==NULL)  node=node->figlio_destro;
            else {
                  tmp = node->sinistro;
                  previous = node;
                  while (tmp->destro!=NULL)
                     {
                       previous = tmp;
                       tmp = tmp->destro;
                     }
                  node->dati = tmp->dati;
                  if (previous==node)    previous->sinistro = tmp->sinistro;
                  else previous->destro = tmp->sinistro;
                }
       if (ptr==*radice)      *radice = node;
       else if (prev->figlio_sinistro == ptr)    prev->figlio_sinistro = node;
            else   prev->figlio_destro = node;
       }
   else if (*radice != NULL) printf("%d non e' presente nell'albero\n", item);
        else  printf("\nL'albero e' vuoto!!");
   return;
}
```

$O(h)$

Albero binario di ricerca: cancellazione

Caso 3: cancellazione di un nodo con grado 2

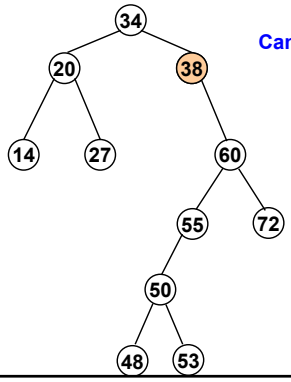


Cancellazione del valore 45

Si fa puntare il figlio dx dell'elemento più grande nel suo sottoalbero sx al sottoalbero dx del nodo da eliminare.

Albero binario di ricerca: cancellazione

Caso 3: cancellazione di un nodo con grado 2



Cancellazione del valore 45

Si fa puntare il figlio dx dell'elemento più grande nel suo sottoalbero sx al sottoalbero dx del nodo da eliminare.

Albero binario di ricerca: cancellazione

Per fusione

```
void delete_nodo_fusione(tree_pointer *radice, int item)
{
 tree_pointer tmp, node, prev=NULL, ptr=*radice;

 while (ptr!=NULL && item!=ptr->dati)
 {
   prev = ptr;
   if(item>ptr->dati)     ptr = ptr->figlio_destro;
   else  ptr = ptr->figlio_sinistro;
 }
 node = ptr;
 if (ptr!=NULL &&  item==ptr->dati) {
      if (node->figlio_destro==NULL)       node = node->figlio_sinistro;
      else if (node->figlio_sinistro==NULL)  node=node->figlio_destro;
            else {
                  tmp = node->figlio_sinistro;
                  while (tmp->figlio_destro!=NULL)    tmp = tmp->figlio_destro;
                  tmp->figlio_destro = node->figlio_destro;
                  node = node->figlio_sinistro;
                }
       if (ptr==*radice)  *radice = node;
       else if (prev->figlio_sinistro == ptr)    prev->figlio_sinistro = node;
            else  prev->figlio_destro = node;
       }
   else if (*radice != NULL) printf("%d non e' presente nell'albero\n", item);
        else  printf("\nL'albero e' vuoto!!");
   return;
}
```

$O(h)$