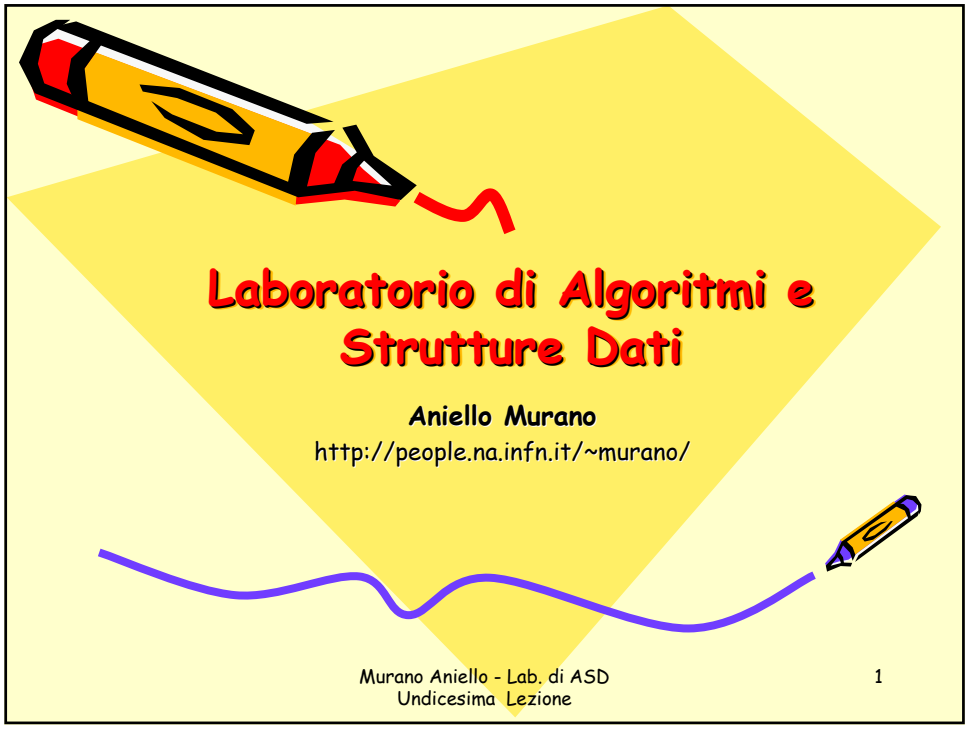
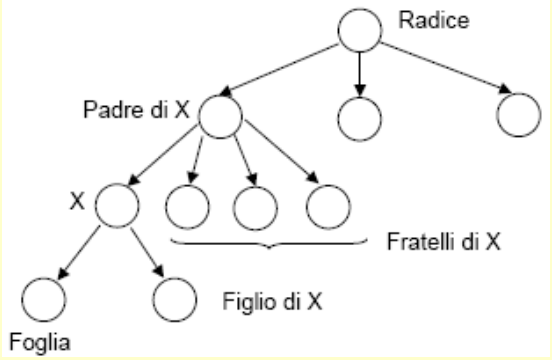

Laboratorio di Algoritmi e Strutture Dati
Aniello Murano
http://people.na.infn.it/~murano/
Murano Aniello - Lab. di ASD
Undicesima Lezione
1


Alberi binari di ricerca
Murano Aniello - Lab. di ASD
Undicesima Lezione
2

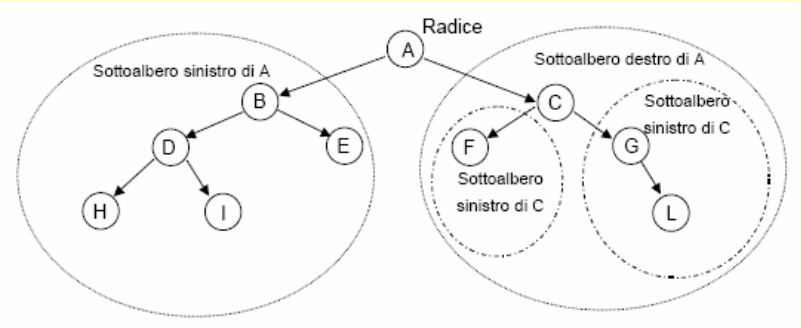
Alberi

- L'albero è un tipo astratto di dato utilizzato per rappresentare relazioni gerarchiche tra oggetti.



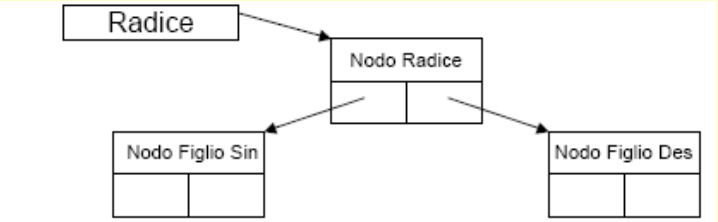
Alberi binari

- Un albero binario è una tipo astratto di dato che o è vuoto (cioè ha un insieme vuoto di nodi) o è formato da un nodo A (detto la radice) e da due sottoalberi, che sono a loro volta alberi binari, e che sono chiamati rispettivamente sottoalbero sinistro e sottoalbero destro.



Rappresentazione di un albero binario

- Per rappresentare un albero binario si puo usare la seguente struttura ricorsiva:




```
struct nodo { int inforadice;
              struct nodo *sinistro,*destro;
            };
struct nodo *radice;
```


Primitive sugli alberi binari

- Per controllare se un nodo è vuoto possiamo usare il seguente codice

```
int test_albero_vuoto (struct nodo *radice)
{ if(radice) return 0;
else return 1; }
```

- Per sapere il valore di un nodo possiamo usare la seguente funzione che ritorna 0 se l'albero è vuoto e memorizza nella variabile valore passata per indirizzo il valore del nodo

```
int  radice (struct nodo *rad, *valore)
{ if(test_albero_vuoto(rad) return 0;
*valore=rad->inforadice;
return 1; }
```

Altre Primitive

- Per avere il puntatore al figlio sinistro (destro) di un nodo:

```
struct nodo * sinistro (struct nodo *rad)
{ if(!test_albero_vuoto(rad)
return rad->sinistro;
else return NULL; }
```

- Per costruire un nodo:

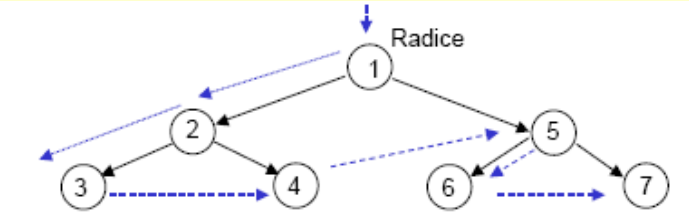
```
struct nodo * costruisci(struct nodo *s, int r, struct nodo *d)
{ struct nodo *aux;
   aux=malloc(sizeof(struct nodo));
   if (aux) { aux->inforadice=r; aux->sinistro=s; aux->destro=d;  }
   return aux;}
```


Visita di un albero binario

- Oltre alle operazioni primitive, si definiscono delle operazioni di visita ovvero di analisi dei nodi di un albero in deteminato ordine.
- Di seguito analizziamo le seguenti visite di un albero:
  - Visita in Preordine
  - Visita in Ordine
  - Visita in Postordine

Visita in Preordine

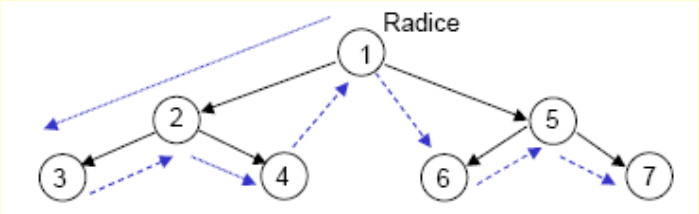
- Nella visita in Preordine, se l'albero non è vuoto:
  - Si analizza la radice dell'albero;
  - Si visita in preordine il sottoalbero sinistro;
  - Si visita in preordine il sottoalbero destro.



- Nella visita in preordine del precedente albero i nodi verrebbero visitati nel seguente ordine: 1, 2, 3, 4, 5, 6, 7

Visita in Ordine

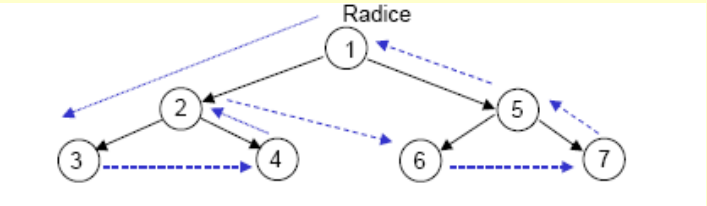
- Nella visita in ordine, se l'albero non è vuoto:
  - Si visita in ordine il sottoalbero sinistro;
  - Si analizza la radice dell'albero;
  - Si visita in ordine il sottoalbero destro.



- Nella visita in ordine del precedente albero i nodi verrebbero visitati nel seguente ordine: 3, 2, 4, 1, 6, 5, 7

Visita Postordine

- **Nella visita in postordine, se** l'albero non è vuoto:
  - Si visita in postordine il sottoalbero sinistro;
  - Si visita in postordine il sottoalbero destro;
  - Si analizza la radice dell'albero.



- Nella visita in ordine del precedente albero i nodi verrebbero visitati nel seguente ordine: 3, 4, 2, 6, 7, 5, 1

Codice per la visita di un albero

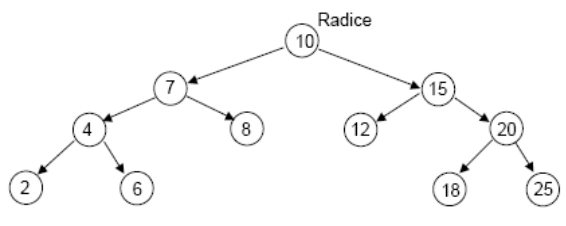
```
void visita_in_preordine(struct nodo *radice) {
    if(radice) {
          printf("%d ",radice->inforadice);
          visita_in_preordine(radice->sinistro);
          visita_in_preordine(radice->destro); } }

void visita_in_ordine(struct nodo *radice){
    if(radice) {
          visita_in_ordine(radice->sinistro);
          printf("%d ",radice->inforadice);
          visita_in_ordine(radice->destro); } }

void visita_in_postordine(struct nodo *radice) {
    if(radice) {
          visita_in_postordine(radice->sinistro);
          visita_in_postordine(radice->destro);
          printf("%d ",radice->inforadice); } }
```

Alberi binari di ricerca

- Un albero binario di ricerca è un albero binario in cui per ogni nodo dell'albero N tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale di quello di N e tutti i nodi del sottoalbero destro hanno un valore maggiore di quello del nodo N.

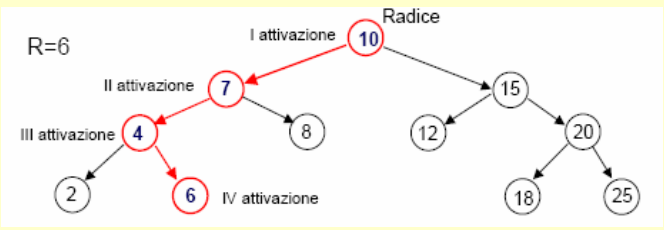


- Il vantaggio principale di tale organizzazione è nella **ricerca**.
- Ogni volta che bisogna ricercare un elemento, il confronto del valore di un nodo dell'albero permette di eliminare dalla fase di ricerca o il sottoalbero corrente di destra o quello di sinistra.

Ricerca in un albero binario di ricerca

- Per trovare un numero R si procede nel seguente modo:
  1. Se l'albero è vuoto l'elemento non è presente;
  2. Se la radice dell'albero == R l'elemento è stato trovato;
  3. Se la radice dell'albero > R la ricerca viene condotta nel sottoalbero sinistro;
  4. Altrimenti la ricerca viene condotta nel sottoalbero destro;

- La ricerca può essere realizzata mediante una funzione ricorsiva che nei casi 3 e 4 invoca se stessa.



Ricerca in un albero binario di ricerca

- Versione iterativa della ricerca

```
int ricerca (struct nodo *radice, int r)
{
   if(test_albero_vuoto (radice) )
            return 0; /* Non Trovato poiché l'albero è vuoto */
   while(radice) {
                  if(radice->inforadice==r)
                        return 1; /* Trovato */
                  else if(radice->inforadice > r)
                        /* Cerca nel sottoalbero sinistro */
                        radice=radice->sinistro;
                     else /* Cerca nel sottoalbero destro */
                        radice=radice->destro;
               }
   return 0;
}
```

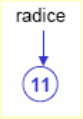

Ricerca in un albero binario di ricerca

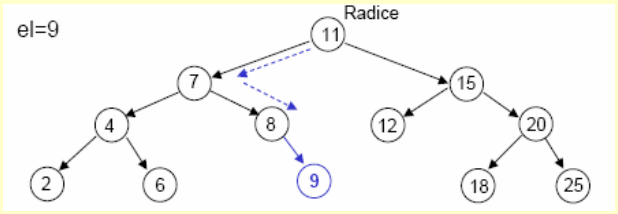
- Versione ricorsiva della ricerca

```
int ricerca (struct nodo *radice,int r)
{
   if(test_albero_vuoto (radice) )
            return 0; /* Non Trovato poiché l'albero è vuoto */
   if(radice->inforadice==r)
            return 1; /* Trovato */
   else if(radice->inforadice > r)
            /* Cerca nel sottoalbero sinistro */
            return ricerca(radice->sinistro,r);
         else /* Cerca nel sottoalbero destro */
            ricerca(radice->destro,r);
}
```

Inserimento di un nuovo nodo in un ABR

- Se l'albero è vuoto, viene creato un nuovo nodo;
- Se l'elemento è minore o uguale alla radice dell'albero, l'inserimento va fatto nel sottoalbero sinistro;
- Se l'elemento è maggiore o uguale alla radice dell'albero, l'inserimento va fatto nel sottoalbero destro;





Inserimento di un nuovo nodo in un ABR

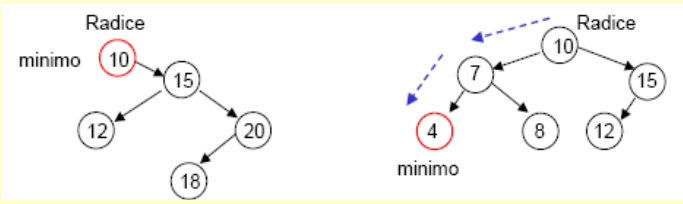
```
struct nodo *inserisci (struct nodo *radice, int e)
    {struct nodo *aux;
    if(radice==NULL)
          /* Creazione di un nuovo nodo */
          {aux=malloc(sizeof(struct nodo));
          if(aux)
                    {aux->info=e;
                    aux->sinistro=aux->destro=NULL;
                    return aux;
                    }
          else printf("Memoria non allocata");
          }
    else if(e<radice->info) radice->sinistro = inserisci(radice->sinistro,e);
          else if(e>radice->info) radice->destro = inserisci(radice->destro,e);
               /* altrimenti il valore è già nell'albero e non si fa niente */
          return radice;
    }
```

Inserimento di un nuovo nodo tramite l'uso di puntatori a puntatori

```
void inserisci (struct nodo **radice, int e)
{   struct nodo *aux;
    if(*radice==NULL)  /* Creazione di un nuovo nodo */
    {       aux=malloc(sizeof(struct nodo));
            if(aux)
            {           aux->info=e;
                        aux->sinistro=aux->destro=NULL;
                        *radice=aux;
            }
            else printf("Memoria non allocata");
    }
    else if((*radice)->info>e)
                        inserisci(&(*radice)->sinistro,e);
         else inserisci(&(*radice)->destro,e);
    return;
}
```


Ricerca minimo in un ABR

- Se il sottoalbero sinistro è vuoto, il minimo è la radice.
- Altrimenti il minimo è da cercare nel sottoalbero sinistro




```
int ricerca_minimo (struct nodo *radice)
{   /* Questa funzione riceve il puntatore alla radice */
    if(radice->sinistro==NULL)
      return radice->inforadice;
    else return ricerca_minimo(radice->sinistro);
}
```

Cancellazione

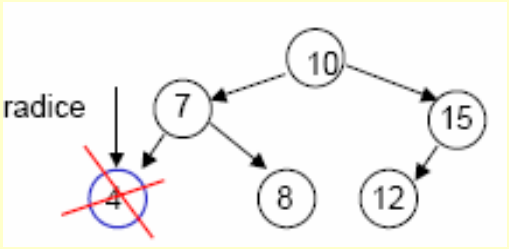
- Nella cancellazione di un elemento ***el*** da un albero bisogna distinguere i seguenti casi:
  1. Albero vuoto: non viene realizzata alcuna cancellazione;
  2. L'elemento el < radice albero: la cancellazione va effettuata nel sottoalbero sinistro;

     elimina(radice->sinistro,el);
  3. L'elemento el > radice albero: la cancellazione va effettuata nel sottoalbero destro;

     elimina(radice->destro,el);
  4. L'elemento el==radice albero. Si considerano i seguenti casi:
     1. La radice è una foglia:
     2. Il nodo ha un sottoalbero non vuoto e l'altro vuoto.
     3. Il nodo ha entrambi i sottoalberi non vuoti.

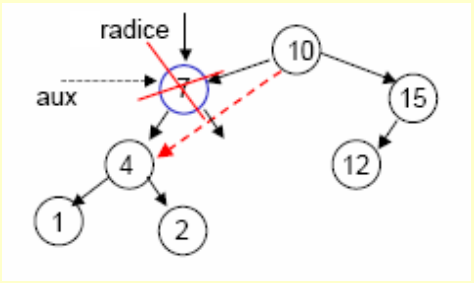
Primo caso

- La radice è una foglia: viene eliminata liberando la memoria del nodo radice



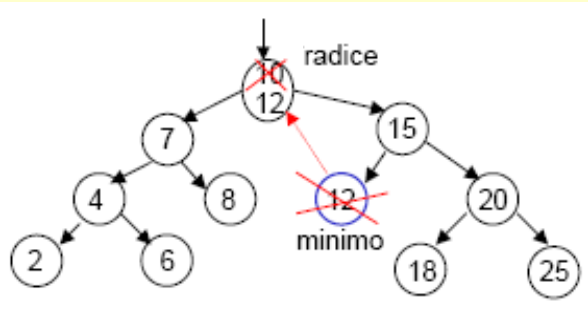
Secondo Caso

- Se il sottoalbero destro e' vuoto:
  - il nodo figlio del nodo padre (10) di el (7) diventa il nodo figlio di el (4);
  - Viene eliminato il nodo el (7).



Terzo Caso

- Il sottoalbero destro e sinistro del nodo el sono non vuoti:
  - viene ricavato il minimo (12) del sottoalbero destro;
  - il minimo (12) viene copiato nel nodo el (10);
  - viene eliminato il minimo (12) del sottoalbero destro.



Codice per la cancellazione (NO)

```
void *elimina (struct nodo *radice, int el)
{   struct nodo *aux;
    if(test_albero_vuoto(radice)) { printf("elemento non trovato");return; }
    if(radice->inforadice > el) /* L'elemento va cercato nel sottoalbero sinistro */
          elimina(radice->sinistro,el);
    else if(radice->inforadice< el) /* L'elemento va cercato nel sottoalb. destro */
          elimina(radice->destro,el);
    else /* Trovato l'e'elemento da cancellare */
          {  if(radice->sinistro && radice->destro) /* Il nodo ha entrambi i figli*/
                    {          aux=ricerca_minimo(radice->destro);
                               radice->inforadice=aux->inforadice;
                               elimina(radice->destro,aux->inforadice); }
             else {    /* Il nodo ha 0 oppure un figlio*/
                    aux=radice;
                    if (radice->sinistro = NULL) radice=radice->destro;
                    else if (radice->destro = NULL) radice=radice->sinistro;
                    free(aux); } }
     return; }
```


Osservazioni

- Il codice precedente, non cancella efficacemente un nodo perchè all'atto della cancellazione "free(radice); radice=NULL;" deallochiamo effettivamente la memoria per il nodo da cancellare, ma il nodo padre di quest'ultimo continuerà a puntare ad una locazione di memoria reale, che non è NULL.
- La funzione non è corretta perchè porta troppo avanti il puntatore, per cui non abbiamo più la possibilità di modificare il campo destro o sinistro del nodo padre.
- Una soluzione è forzare la modifica del nodo padre utilizzando una variabile di comodo.
- Un'altra soluzione è quella di utilizzare il metodo del puntatore a puntatore.
- Di seguito mostriamo entrambe le soluzioni.

Codice per la cancellazione con variabile di comodo "foglia"

```
void *elimina (struct nodo *radice, int el)
{   struct nodo *aux;
    if(test_albero_vuoto(radice)) { printf("elemento non trovato");return; }
    if(radice->inforadice > el) /* L'elemento va cercato nel sottoalbero sinistro */
          { elimina(radice->sinistro,el);
          if (foglia) {radice->sinistra=NULL; foglia=0;}}
    else if(radice->inforadice< el) /* L'elemento va cercato nel sottoalb. destro */
          { elimina(radice->destro,el);
            if (foglia) {radice->destro=NULL; foglia=0;}}
    else /* Trovato l'e'elemento da cancellare */
          {  if(radice->sinistro && radice->destro) /* Il nodo ha entrambi i figli*/
                    {          aux=ricerca_minimo(radice->destro);
                               radice->inforadice=aux->inforadice;
                               elimina(radice->destro,aux->inforadice);
                               if (foglia) {radice->sinistra=NULL; foglia=0;}}
             else {    /* Il nodo ha 0 oppure un figlio*/
                    aux=radice;
                    if(!radice->sinistro && !radice->destro) foglia=1;
                    if (radice->sinistro = NULL) radice=radice->destro;
                    else if (radice->destro = NULL) radice=radice->sinistro;
                    free(aux); } }
    return; }
```


Uso di puntatori a puntatori

```
void elimina (struct nodo **radice, int el)
{ struct nodo *aux;
  int minimo;

  if(test_albero_vuoto(*radice)) return;

  if((*radice)->inforadice> el)
        /* L'elemento da eliminare va cercato
           nel sottoalbero sinistro */
       elimina(&(*radice)->sinistro,el);
  else if((*radice)->inforadice< el)
        /* L'elemento da eliminare va cercato
           nel sottoalbero destro */
       elimina(&(*radice)->destro,el);
  else
     {if((*radice)->sinistro==NULL && (*radice)->destro==NULL)
        /* L'elemento da eliminare e' una foglia */
          { free(*radice);
            *radice=NULL;
            return;
          }
```

Uso di puntatori a puntatori

```
        /* Se il sottoalbero destro e' vuoto */
         if((*radice)->destro==NULL)
            { aux=*radice;
              *radice=aux->sinistro;
              free(aux);
              return;
            }
        /* Se il sottoalbero sinistro è vuoto */
           if((*radice)->sinistro==NULL)
            { aux=*radice;
              *radice=aux->destro;
              free(aux);
              return;
            }
         minimo=ricerca_minimo((*radice)->destro);
        (*radice)->info=minimo;
        elimina(&(*radice)->destro,minimo);
      }
}
```