1 Alberi

L'albero ordinato (spesso chiamato più semplicemente "albero") è una struttura informativa fondamentale, che si presta a rappresentare svariate situazioni, quali:

- Partizioni successive di un insieme in sottoinsiemi disgiunti;
- Organizzazioni gerarchiche di dati;
- Procedimenti enumerativi o decisionali.

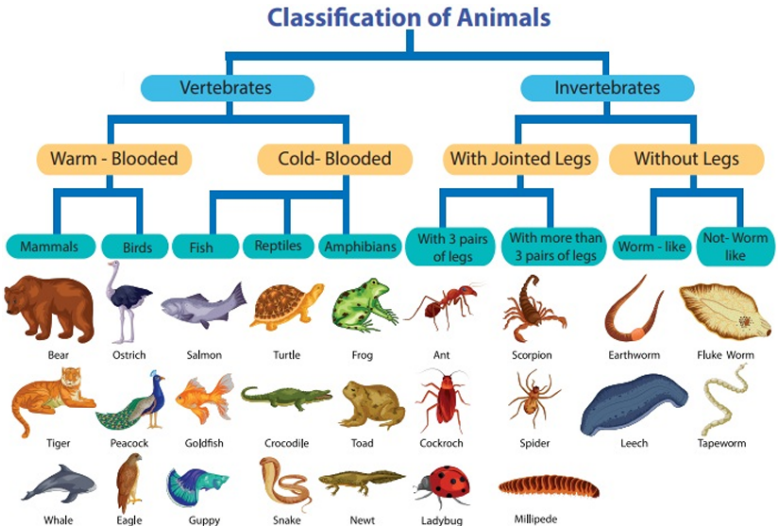
L'**implementazione** avviene, come per le liste, attraverso l'uso di **strutture autoreferenziali** e **puntatori**.

Struttura di un albero ordinato (o radicato)

Un **albero ordinato** è dato da un insieme finito di elementi (dunque una **struttura dati statica**) detti **nodi** e un insieme di **archi orientati** che connettono coppie di nodi.

Ogni albero presenta le seguenti proprietà:

- Un nodo dell'albero è designato come nodo **radice**;
- Ogni nodo n, a parte la radice, ha esattamente un **arco entrante**;
- Esiste un **cammino unico** dalla radice a ogni nodo;
- L'albero **è connesso**: non esistono elementi che non fanno parte dell'albero, ovvero che non sono connessi tramite il cammino di archi all'albero stesso.



Dunque, l'albero rappresenta una struttura gerarchica nella quale a partire da un elemento molto grande (come in questo cao il "regno animale") si divide in sottoclassi, fino ad arrivare ai singoli elementi.

Un esempio che si avvicina maggiormente all'ambito dei sistemi operativi, come si può vedere in Figure 22, è un albero del file system (come quello di linux), dove si ha un'organizzazione in cartelle, partendo dalla radice, per ogni elemento dell'albero, fino ad arrivare ai singoli file.

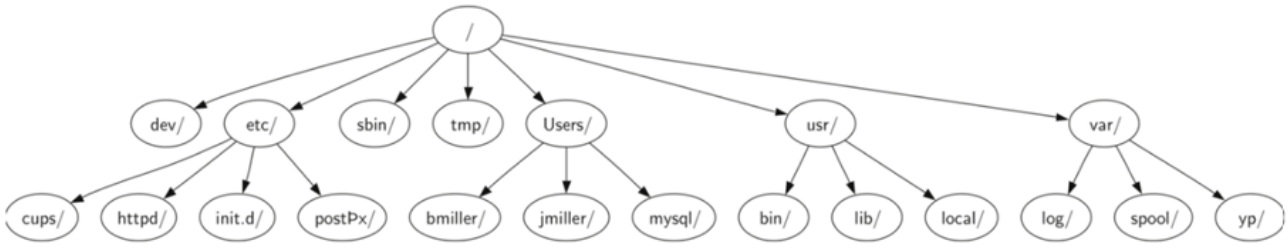


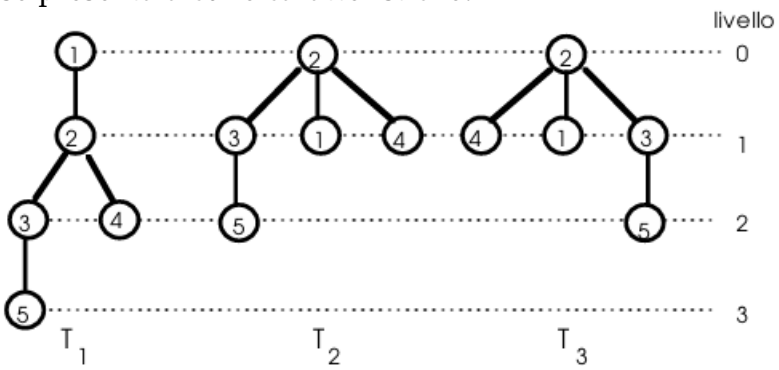
Figure 22: Albero del file system linux

1.1 Terminologia

Sia T un albero ordinato di n nodi, con radice r.
Possiamo definire $T_1, ..., T_k$ gli insiemi disgiunti e non vuoti in cui sono partizionati tutti i nodi di T, ognuno dei quali avrà come radice un nodo $r_1, ..., r_k$.

- Ciascun T_i è detto **sottoalbero** di T, mentre ciascun r_i è detto **figlio** di r;
- I nodi $r_1, ..., r_k$ sono tra loro **fratelli**, ed r è il loro **padre**.
- Inoltre, un nodo senza figli è detto **foglia**, mentre la radice dell'abero (**root**) è l'unico **nodo senza padre**.

Oltre alla terminologia che assumono i nodi dell'albero in base alla loro posizione, l'albero in se presenta alcune caratteristiche:

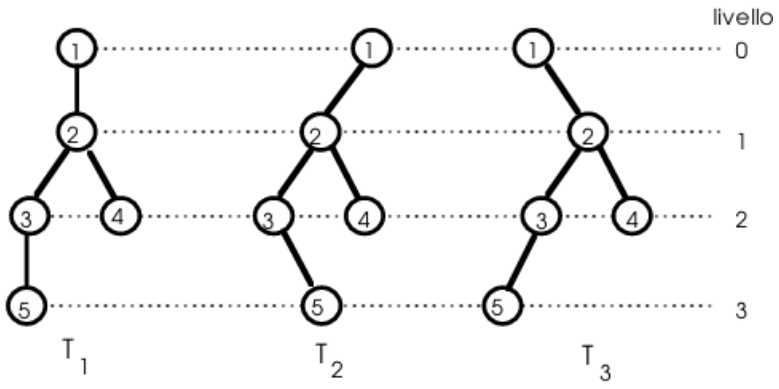


- ***Profondita dei nodi (depth)***: profondità del cammino semplice dalla radice al nodo (misurata in numero di archi percorsi). Ad esempio, la radice avrà profondità 0, i figli avranno profondità 1, i figli dei figli avranno profondità 2 e cosi via...
- ***Altezza albero (height)***: indica la profondità massima delle sue foglie. Ad esempio, T_1 ha altezza 3, mentre T_2 e T_3 altezza 2;
- ***Livello (level)***: si riferisce all'insieme di nodi alla stessa profondità;

1.2 Alberi binari

Cos'è un'albero binario?

Un albero binario è un particolare albero ordinato in cui ogni nodo ha **al più due figli**, e si fa distinzione tra figlio **sinistro** e figlio **destro**.
N.B: in alcune implementazioni può esserci un puntatore al padre.



La proprietà degli alberi binari è **molto delicata**: infatti, anche se due alberi T_1 e T_2 hanno gli stessi nodi e la stessa radice, essi risultano diversi se in T_1 un nodo u è figlio sinistro di un nodo v, mentre in T_2 lo stesso nodo u è figlio destro di v.

Nell'esempio mostrato, l'abero T_1 presenta al massimo due figli per ogni nodo, ma la rappresentazione grafica non aiuta a comprendere se si tratta di un figlio destro o sinistro. Al contrario T_2 e T_3 sono alberi binari e distinti, poiché non hanno gli stessi figli destri e sinistri.

1.2.1 Creazione di un albero binario

Come per qualsiasi struttura dati (liste concatenate, doppiamente concatenate, ecc...) anche per poter utilizzare un albero è necessario definire una struttura e inizializzarla nel main:
Struttura dell'albero

```
struct inttree {
int data;
struct inttree *left, *right;
};
```

Inizializzazione dell'albero

```
inttree *root = NULL;
//Puntatore alla struttura
```

Per quanto riguarda la struttura dell'albero:
- `data`: contiene il valore del nodo;
- `*left`: è il puntatore al nodo sinistro;
- `*right`: è il puntatore al nodo destro.

Invece, l'inizializzazione dell'albero è effettuata tramite, `root` rappresenta la radice che all'inizio è impostata a `null` per indicare l'albero vuoto.

Alberi binari e definizione ricorsiva

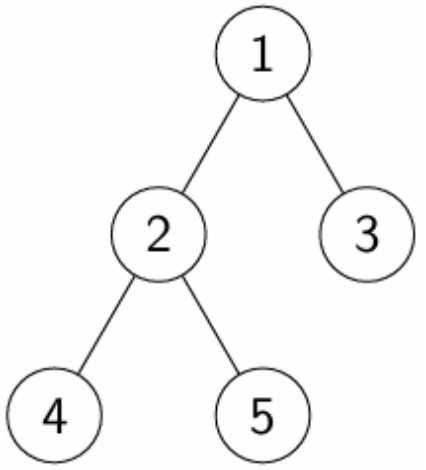
Gli alberi binari ammettono una **definizione ricorsiva** come insieme finito di nodi, che può essere:
- Un **insieme vuoto**, cioè nessun nodo.
  Rappresenta l'albero "vuoto" senza radice e senza figli, come quando viene inizializzato nel main e nessun è stato ancora allocato in memoria (`inttree *root = NULL;`)
- Oppure, un nodo radice con **due sottoalberi binari disgiunti**, uno sinistro e uno destro.

1.2.2 Costruzione di un albero binario

Dopo aver proceduto con la creazione della struttura e l'inizializzazione dell'albero, si procede con la sua creazione, allocando i vari nodi e definendo i puntatori per il figlo destro e sinistro.

```
int main(int argc, char *argv[]) {
//Livello 0
inttree *root = NULL;
root = malloc(sizeof(inttree));
root->data = 1;

//Livello 1
root->left = malloc(sizeof(inttree));
root->right = malloc(sizeof(inttree));
root->left->data = 2;
root->right->data = 3;
root->right->left = NULL;
root->right->right = NULL;

//Livello 2
root->left->left = malloc(sizeof(inttree));
root->left->right = malloc(sizeof(inttree));
root->left->left->data = 4;
root->left->right->data = 5;
root->left->left->left = NULL;
```

```
21 root->left->left->right = NULL;
22 root->left->right->left = NULL;
23 root->left->right->right = NULL;
24 }
```



- **Blocco 1**: vengono allocati i nodi al livello 0 dell'albero.
  - *riga 2*: il puntatore `root` (`*root`), definito in precedenza, viene associato al nodo radice dell'albero, puntando a un'area di memoria allocata dinamicamente della dimensione della struttura `inttree`;
  - *riga 3*: inizializza il campo `data` del nodo radice a 1
- **Blocco 2**: vengono allocati i nodi al livello 1 dell'albero.
  - *riga 6-7*: il nodo radice presenta al suo interno due puntatori, `left` e `right`, i quali punteranno a due nodi `inttree` differenti, corrispondenti rispettivamente ai figli sinistro e destro;
  - *riga 8-9*: viene inizializzato il campo `data` dei figli della radice, assegnando 2 al figlio sinistro e 3 al figlio destro;
  - *riga 10-11*: come per il nodo radice, anche le strutture dei suoi nodi figli avranno dei puntatori `left` e `right`. In queste due righe i puntatori `left` e `right` del figlio destro vengono impostati a `NULL`, a indicare che esso non ha ulteriori discendenti.
- **Blocco 3**: vengono allocati i nodi al livello 2 dell'albero.
  - *riga 13-14*: i puntatori `left` e `right` del figlio sinistro della radice vengono associati a due nuovi nodi `inttree`;
  - *riga 15-16*: viene inizializzato il campo `data` di questi due nodi, assegnando 4 al figlio sinistro e 5 al figlio destro;
  - *riga 17-20*: i puntatori `left` e `right` di entrambi questi nodi vengono impostati a `NULL`, indicando che non hanno ulteriori figli.

1.3 Le visite

Cos'è una visita?

Una **visita** (o attraversamento) di un albero ordinato è una strategia consiste nel seguire una "rotta" di viaggio che consenta di **analizzare ogni nodo** dell'albero almeno una volta.

Una visita può essere eseguita in **due modi**:

- **Visita in profondità** (Deep-First Search - **DFS**): in questo caso si segue un percorso radice-foglia, ovvero per visitare un albero **si visita ricorsivamente** ognuno dei suoi **sottoalberi**, andando da sinistra verso destra. Questo tipo di visita presenta **tre varianti** (ordini):
  - Ordine anticipato (**preorder**);
  - Ordine posticipato (**postorder**);
  - Ordine simmetrico (**inorder**).

  Un'analisi di questo tipo **richiede uno stack**: si visita l'albero utilizzando uno stack che però viene già reso disponibile tramite il **meccanismo di ricorsione**.

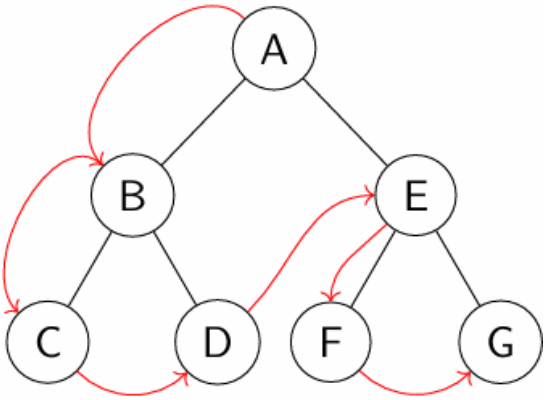
- **Visita in ampiezza** (Breadth First Search - BFS): partendo dalla radice, vengono visitati i nodi livello per livello, "orizzontalmente", fino a raggiungere il livello massimo dell'albero. Proprio per questo, un analisi di questo tipo viene anche detta ***ordine per livelli***.

Sia T un albero non vuoto di radice r. Se r non è una foglia ma ha $k > 0$ figli, indichiamo con $T_1, ..., T_k$ i k sottoalberi di T radicati nei k figli di r. Gli ordini di visita in profondità sono definiti ricorsivamente come segue:

1.3.1 Visita in preorder (DFS)

Funzionamento della visita in preorder

La **visita in preorder** di T consiste nell'esaminare r e poi nell'effettuare, nell'ordine, la previsita dei sottoalberi $T_1, ..., T_k$.



(a) albero attraversato in preorder

```
dfs(Tree t)
1: if t ≠ nil then
2:     % pre-order visit of t
3:     print t
4:
5:     dfs(t.left())
6:
7:     dfs(t.right())
8: end if
```

(b) Pseudocodice visita in preorder

Per capire il funzionamento della visita in preorder, dividiamo il procedimento in alcuni passi, vedendo come evolvolvono la sequenza di stampa (attraversamento dell'albero), e lo stack di chiamate (ricorsività della funzione `dfs()`):

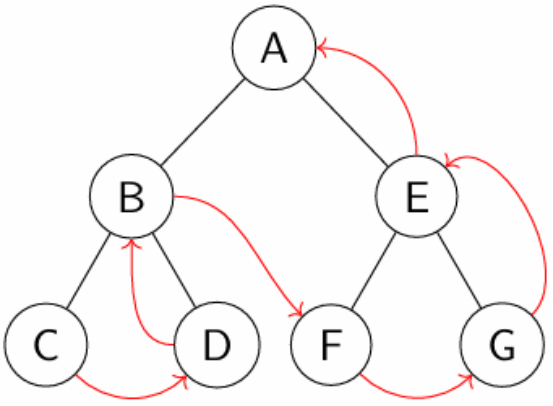
1. In questo tipo di visita si parte sempre chiamando la funzione rispetto al nodo $A \rightarrow$ `dfs(Tree A)`. Pertanto la sequenza di stampa inizia con A, e la chiamata viene memorizzata nello stack delle chiamate.
   - *Sequenza di stampa*: A
   - *Stack chiamate*: A
2. Dopo aver stampato il nodo A, la funzione richiama ricorsivamente `dfs` sul figlio sinistro del nodo specificato nei parametri della funzione `dfs(Tree A)`: succede quindi che la funzione viene rieseguita dall'inizio, usando come riferimento il nodo B, che verrà stampato.
   - *Sequenza di stampa*: $A - B$
   - *Stack chiamate*: $A - B$
3. Analogamente, dopo aver stampato il nodo B, la funzione richiama ricorsivamente la `dfs` sul figlio sinistro del nodo specificato nei parametri della funzione `dfs(Tree B)`, rieseguendo la funzione con C come riferimento.
   - *Sequenza di stampa*: $A - B - C$
   - *Stack chiamate*: $A - B - C$
4. A questo punto, viene chiamata `dfs(t.left())` sul nodo C, ma quest'ultimo è `NULL` poiché non presenta figli; la stessa cosa succede provando a chiamare `dfs(t.right())`.
   Quando entrambe le chiamate ai figli sinistro e destro terminano, significa che il nodo

in questione ha completato la propria porzione di codice ricorsivo e può andare avanti: andare avanti significa che la chiamata `dfs(t.left())` eseguita per `dfs(Tree B)` è stata completata.
   - *Sequenza di stampa*: $A - B - C$
   - *Stack chiamate*: $A - B$
5. Il controllo torna al nodo B che, dopo aver completato la chiamata per il figlio sinistro, eseguirà la chiamata per il figlio destro (D) tramite `dfs(t.right())`.
   - *Sequenza di stampa*: $A - B - C$
   - *Stack chiamate*: $A - B - D$
6. Con la chiamata ricorsva sul figlio destro di B, la funzione viene rieseguita con D come riferimento `dfs(Tree D)`, e come prima operazione D viene stampato.
   - *Sequenza di stampa*: $A - B - C - D$
   - *Stack chiamate*: $A - B - D$
7. Come nel caso del nodo C, anche D non presenta ulteriori figli, dunque le sue chiamate falliranno (`NULL`) e di conseguenza D completa la propria porzione di codice: il controllo torna nuovamente a B.
   - *Sequenza di stampa*: $A - B - C - D$
   - *Stack chiamate*: $A - B$
8. Nonostante il controllo sia tornato a B, esso ha completato le sue chiamate per i figli sinistro e destro, quindi ancora una volta il controllo passa al nodo superiore (in questo caso la radice r (A)).
   - *Sequenza di stampa*: $A - B - C - D$
   - *Stack chiamate*: A
9. Ciò che succede è che in A la chiamata `dfs(t.left())` è stata completata (tutto il sottoalbero sinistro), quindi procedo con la chiama ricorsiva per il nodo destro della radice E: in questo modo **il procedimento si ripete in modo analogo** per tutto il sottoalbero destro.
   - *Sequenza di stampa*: $A - B - C - D - E - \dots$
   - *Stack chiamate*: $A - E - \dots$

1.3.2 Visita in postorder (DFS)

Funzionamento della visita in postorder

La **visita in postorder** di T consiste nell'effettuare, nell'ordine, la postvisita di $T_1, ..., T_k$ e poi nell'esaminare r.



(a) albero attraversato in postorder

```
dfs(Tree t)
1: if t ≠ nil then
2:     dfs(t.left())
3:
4:     dfs(t.right())
5:
6:     % post-order visit of t
7:     print t
8: end if
```

(b) Pseudocodice visita in postorder

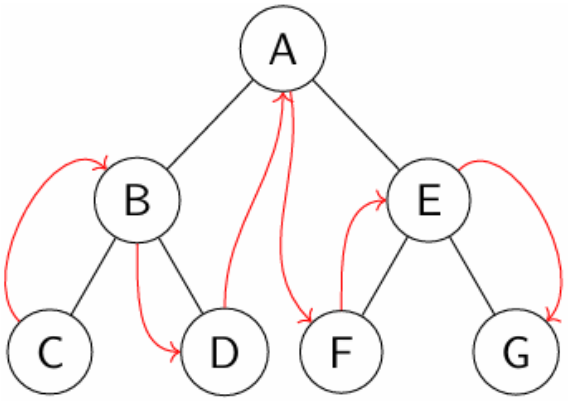
Per quanto riguarda la visita in postorder, anche in questo caso, dividiamo il procedimento in alcuni passi, monitorando l'evoluzione della sequenza di stampa e dello stack di chiamate:

1. Si parte come sempre dal nodo radice $A \rightarrow$ `dfs(Tree A)`. Tuttavia, in postorder la **radice non viene stampata subito**: la funzione procede prima a richiamare `dfs(t.left())`, dunque `dfs()` viene eseguita in modo ricorsivo rispetto al figlio sinistro (B).
   - **Sequenza di stampa**: $-$
   - **Stack chiamate**: A
2. Essendo nuovamente all'inizio della funzione, B non viene stampato, ma ancora una volta viene richiamata `dfs(t.left())` in modo ricorsivo per il figlio sinistro di B (Ovvero C)
   - **Sequenza di stampa**: $-$
   - **Stack chiamate**: $A - B$
3. Il figlio sinistro di B è il nodo C, dunque verrà eseguita la funzione `dfs(Tree C)`. A questo punto la funzione riparte dall'inizio, ma C non ha ulteriori figli, quindi `dfs(t.left())` e `dfs(t.right())` restituiscono `NULL` e si passa alla stampa del nodo (C).
   - **Sequenza di stampa**: C
   - **Stack chiamate**: $A - B - C$
4. Il nodo C, dopo la stampa, ha completato la propria porzione di codice, dunque il controllo ritorna al nodo B.
   - **Sequenza di stampa**: C
   - **Stack chiamate**: $A - B$
5. Il nodo B, dopo aver completato la chiamata per il figlio sinistro, può procedere a richiamare la funzione ricorsiva per il figlio destro `dfs(t.right())`, ovvero il nodo D: ciò implica `dfs(Tree D)`.
   - **Sequenza di stampa**: C
   - **Stack chiamate**: $A - B - D$
6. Anche il nodo D non presenta figli, dunque `dfs(t.left())` e `dfs(t.right())` restituiscono `NULL` e si passa alla stampa del nodo stesso, con conseguente terminazione della sua porzione di codice. Infine, il controllo ritorna al nodo B.
   - **Sequenza di stampa**: $C - D$
   - **Stack chiamate**: $A - B$
7. A questo punto, il nodo B ha completato la visita dei propri figli sinistro (C) e destro (D). Di conseguenza si va oltre le chiamate `dfs(t.left())` e `dfs(t.right())` e si stampa B. Dopo la stampa di B, si ha la terminazione della sua porzione di codice ricorsivo, e il controllo ritorna al nodo A.
   - **Sequenza di stampa**: $C - D - B$
   - **Stack chiamate**: A
8. Terminata la visita del sottoalbero sinistro di A, la funzione procede ora con il figlio destro, eseguendo `dfs(t.right())` e quindi in modo ricorsivo la funzione `dfs(Tree E)`.
   - **Sequenza di stampa**: $C - D - B$
   - **Stack chiamate**: $A - E$
9. Dunque, una volta che `dfs(Tree E)` incomincia le sue chiamate ricorsive, viene **visitato tutto il sottoalbero destro**, con un **procedimento analogo** a quello utilizzato precedentemente per visitare il sottoalbero sinistro.
   - **Sequenza di stampa**: $C - D - B - \ldots$
   - **Stack chiamate**: $A - E - \ldots$

1.3.3 Visita inorder (DFS)

Funzionamento della visita inorder

Fissato $i \geqslant 1$, la **visita inorder** di T consiste nell'effettuare la invista di $T_1, ..., T_i$, nell'esaminare r, e poi nell'effettuare la invisita di $T(i+1), ..., T_k$.



(a) albero attraversato inorder

```
dfs(Tree t)
1: if t ≠ nil then
2:     dfs(t.left())
3:
4:     % in-order visit of t
5:     print t
6:
7:     dfs(t.right())
8: end if
```

(b) Pseudocodice visita inorder

Anche in questo caso analizziamo passo per passo il funzionamento della funzione `dfs()` e l'evoluzione della sequenza di stampa e dello stack di chiamate:

1. La visita incomincia chiamando `dfs(Tree A)` sul nodo radice (A). Come per la visita in postorder, la radice non viene subito stampata, ma si procede a richiamare a richiamare ricorsivamente il suo figlio sinistro (B) tramite `dfs(t.left())`.
   - **Sequenza di stampa**: $-$
   - **Stack chiamate**: A
2. Quando la funzione `dfs(Tree B)` viene richiamata ricorsivamente sul nodo B, si riparte dall'inizio e nuovamente si chiama in modo ricorsivo il figlio sinistro di B (C).
   - **Sequenza di stampa**: $-$
   - **Stack chiamate**: $A - B$
3. Una volta che `dfs(Tree C)` è stata chiamata, la chiamata ricorsiva al figlio sinistro di C, all'interno della funzione, restituisce `NULL` proprio perché il nodo C non ha figli.
   Dunque si procede con l'operazione subito successiva, la stampa del nodo, per poi trovare un'altra chiamata ricorsiva al figlio destro di C che restituià anch'essa `NULL` per lo stesso motivo di prima.
   - **Sequenza di stampa**: C
   - **Stack chiamate**: $A - B - C$
4. Arrivati a questo punto la porzione di codice eseguita da C termina e il controllo torna al nodo B.
   - **Sequenza di stampa**: C
   - **Stack chiamate**: $A - B$
5. Quando il nodo B termina la chiamata al figlio sinistro (C), l'operazione successiva è la stampa del nodo stesso. Subito dopo la stampa viene eseguita una chiamata ricorsiva al figlio destro (nodo D).
   - **Sequenza di stampa**: $C - B$
   - **Stack chiamate**: $A - B$

6. Non appena `dfs(Tree D)` è in esecuzione vengono eseguite le istruzioni secondo l'ordine prestabilito: `dfs(t.left())` restituisce `NULL` (nessun figlio sinistro), D viene stampato e `dfs(t.right())`, non avendo figli, restituisce anch'esso `NULL`.
   - **Sequenza di stampa**: $C - B - D$
   - **Stack chiamate**: $A - B - D$
7. Dopo l'esecuzione di tutta la porzione di codice del nodo D, il controllo torna a B, ma anche quest'ultimo, avendo visitato figlio sinistro e destro, ha terminato le istruzioni nella sua porzione di codice. Quindi, in cascata, il controllo torna al nodo A, che, dopo aver visitato tutto il suo sottoalbero sinistro tramite la chiamata ricorsiva `dfs(t.left())` può essere stampato come prevede l'istruzione successiva della sua porzione di codice.
   - **Sequenza di stampa**: $C - B - D - A$
   - **Stack chiamate**: A
8. Infine, dopo la stampa del nodo A, viene effettuata una chiamata al sottoalbero destro tramite `dfs(t.right())`. Anche in questo caso, **il procedimento di visita è analogo** a quello utilizzato precedentemente per il **sottoalbero sinistro**.
   - **Sequenza di stampa**: $C - B - D - A - \dots$
   - **Stack chiamate**: $A - \dots$

1.3.4 Attraversamento di un albero binario in C

La funzione per l'attraversamento di un albero binario fa riferimento alla visita in preorder.

```
void preorder(inttree *p, void(*op)(inttree*)){
if(p){
    (*op)(p);
    preorder(p->left,op);
    preorder(p->right,op);
}
}
```

- **Tipo di ritorno**: la funzione non restituisce alcun valore, infatti il suo scopo è unicamente quello di visitare i nodi dell'albero;
- **Parametri**: la funzione `preorder` accetta due parametri;
  - `inttree *p`: è un puntatore alla struttura del nodo di un albero, definita al capitolo 1.2.1. Solitamente viene passato il puntatore alla radice del nodo dell'albero, `*root`, definito nel main, per poter iniziare la visita di tutto l'albero. Successivamente, tramite le chiamate ricorsive, il puntatore non farà più riferimento alla radice ma ai vari nodi figli;
  - `void (*op) (inttree*)`: è un puntatore a funzione (`(*op)`), **cioè una funzione passata come argomento**, che accetta come parametro un puntatore alla struttura di un nodo dell'albero (`inttree*`) e non restituisce nulla (`void`).
    In questo modo, passando come argomento a `preorder` il nome di una qualsiasi funzione che come argomento ha un puntatore alla struttura di un nodo dell'albero, è possibile eseguire un'operazione specifica all'interno della visita.

    Ad esempio, può essere utilizzato per richiamare una funzione che si occupa della stampa dei nodi, così da **tenere traccia della sequenza di attraversamento**.

```
void stampaNodo(inttree *n) {
printf("%d ", n->data);
```

```
}
```

- **Funzionamento del codice**: il corpo della funzione presenta un controllo condizionale e due chiamate ricorsive per l'esecuzione della visita dell'albero.
  - *riga 2*: viene effettuato un controllo di validità sul nodo che si sta visitando, assicurandosi che esista (cioè che non sia `NULL`). Questo controllo risulta utile nel momento in cui si raggiunge una foglia, i cui figli sinistro e destro non esistono, per fare in modo di cedere il controllo al nodo successivo;
  - *riga 3*: viene eseguita la funzione passata come parametro; `(*op)` rappresenta la funzione puntata da `op`, mentre `p` è l'argomento (il nodo attualmente visitato) che le viene passato;
  - *riga 4*: viene effettuata una chiamata ricorsiva a `preorder`, specificando come parametri il figlio sinistro (`p->left`) del nodo corrente e il puntatore alla funzione `op`.
    In questo modo, la visita procede ricorsivamente su tutti i figli sinistri dell'albero;
  - *riga 5*: allo stesso modo, quando il figlio sinistro di un determinato nodo è stato visitato, con `p->right` si passa alla visita ricorsiva del figlio destro dello stesso nodo.

N.B: ovviamente per creare le altre tipologie di visite (postorder e inorder), è sufficiente invertire l'ordine delle istruzioni all'interno della condizione `if` secondo la metodologia prevista dal tipo di attraversamento.

Esempio di utilizzo

```
int main(int argc, char *argv[]) {
inttree *root = NULL;
root = malloc(sizeof(inttree));

// Allocazione di tutti i nodi e assegnazione del valore per ognuno
// come fatto al capitolo 1.2.2.
// ...
// ...

// Esecuzione della visita preorder
printf("Attraversamento in preorder: ");
preorder(root, stampaNodo);
printf("\n");

return 0;
}
```

1.3.5 Cancellazione di un albero

Per poter effettuare la cancellazione di un albero, la seguente implementazione fa leva sull'utilizzo di una struttura ricorsiva, in modo da **liberare correttamente** tutta la memoria occupata da un albero binario.

```
1 void destroytree (inttree *p) {
2 if (p->left){ // Se esiste un sottoalbero sinistro
3     destroytree (p->left); // Libera il sottoalbero
4 }
```

```
5
6  if (p->right){ // Se esiste un sottoalbero destro
7      destroytree (p->right); // Libera il sottoalbero
8  }
9
10 free (p); / Per finire libera la memoria della radice
11 }
```

- **Tipo di ritorno**: la funzione non restituisce alcun valore, infatti in suo scopo è unicamente quello di **rilasciare la memoria** precedentemente allocata per ciascun nodo dell'albero.
- **Parametri**: la funzione `destroytree` accetta un singolo parametro, ovvero `inttree *p`. Quest'ultimo è un puntatore alla struttura del nodo dell'albero, definita al capitolo 1.2.1. A partire dal nodo specificato (in genere la radice `root`), la funzione visita ricorsivamente tutti i sottoalberi (sinistro e destro), liberando la memoria associata a ciscun ogni nodo.
- **Funzionamento del codice**: l'idea di base della funzione `destroy` è che "*non posso liberare un nodo finché non ho liberato i figli, perché se liberassi il nodo prima, perderei il puntatore ai figli e non potrei più raggiungerli per deallocarli*".

  Dunque, la sequenza di cancellazione segue l'ordine di **visita postorder** discussa al capitolo 1.3.2, quindi, considerando lo stesso albero, $C - D - B - F - G - E - A$. In accordo con il funzionamento postorder, il codice presenta la **seguente logica**:
  - *righe 2-4*: viene effettuato un controllo sul figlio sinistro del nodo passato come parametro. Se esiste, la funzione `destroytree` viene richiamata ricorsivamente fino a raggiungere una foglia (nodo privo di figli), che restituirà un controllo falso.
  - *riga 6-8*: quando il controllo su `p->left` risulta falso, si passa al controllo su `p->right`. Anche in questo caso viene richiamata la funzione `destroytree` in modo ricorsivo fino a che non si arriva ad una foglia, che per definizione non presenta figli.
  - *riga 10*: quando entrambi i controlli risultano falsi, significa che il nodo corrente non ha più figli, quindi può essere deallocato in sicurezza tramite `free(p)`.
    Al termine della liberazione, la funzione termina e il controllo ritorna al nodo superiore che era in attesa della conclusione della chiamata ricorsiva nello **stack delle chiamate**.

Esempio di utilizzo

```c
int main(int argc, char *argv[]) {
inttree *root = NULL;
root = malloc(sizeof(inttree));

// Allocazione di tutti i nodi e assegnazione del valore per ognuno
// come fatto al capitolo 1.2.2.
// ...
// ...

// Ora, per liberare tutta la memoria occupata da questo albero, basta
   una sola chiamata
destroytree(root);

return 0;
}
```

1.4 Alberi binari di ricerca (BST)

Come detto precedentemente al capitolo **??** un **dizionario** in informatica è una **struttura dati astratta** che memorizza coppie (*chiave, valore*), quindi servono a memorizzare dati associativi come, ad esempio, una rubrica telefonica o una tabellla di simboli.
I dizionari consentono **tre operazioni fondamentali:**

- `lookup(Item k)` $\rightarrow$ cercare un elemento con chiave `k`;
- `insert(Item k, Item v)` $\rightarrow$ inserire un elemento con chiave `k` e valore `v`;
- `remove(Item k)` $\rightarrow$ rimuovere l'elemento con chiave `k`.

Un dizionario può essere implementato in vari modi:

Struttura dati	`lookup`	`insert`	`remove`
Vettore ordinato	$O(\log n)$	$O(n)$	$O(n)$
Vettore non ordinato	$O(n)$	$O(1)$*	$O(1)$*
Lista non ordinata	$O(n)$	$O(1)$*	$O(1)$*

*=Si assume che l'elemento sia già stato trovato, altrimenti $O(n)$

Queste tipologie di implementazioni presentano diversi **limiti di efficienza**. Ad esempio, i **vettori ordinati** permettono una ricerca veloce ma inserimenti e rimozioni lente, poichè dovrei spostare tutti gli elementi ($O(n)$). Nei **vettori non ordinati**, invece, la ricerca richiede di scorrere tutti gli elementi ($O(n)$), mentre l'inserimento e la rimozione sono operazioni facili e veloci ($O(1)$) che prevedono un quantitativo di operazioni fisso, purché si conosca già la posizione dell'elemento su cui operare, come un'inserimento in testa e una rimozione dalla coda. Se invece fosse necessario inserire o rimuovere in una posizione specifica, il costo salirebbe nuovamente a $O(n)$.
Si conclude quindi che **utilizzare i vettori non è una buona idea**.

L'idea degli alberi binari di ricerca

Per risolvere il problema legato all'inefficienza dei vettori, vengono introdotti gli **alberi binari di ricerca** (BST - Binary Search Trees) proprio come una **struttura dati efficiente** per implementare i dizionari dinamici ordinati.

Cos'è un albero binario di ricerca?

Un **albero binario di ricerca** è un modo per realizzare una **struttura dati** che mantiene **i dati in ordine**, proprio come farebbe un vettore ordinato, grazie alla sua costruzione secondo **una logica *dicotomica***. Inoltre, all'interno dell'albero **ogni nodo rappresenta una coppia** (*chiave, valore*).

La logica dicotomica implica che tra i nodi valgano **le seguenti proprietà**, che, per come sono definite, si applicano **in maniera ricorsiva**:

- Per ogni nodo u, tutte le chiavi contenute nel sottoalbero radicato nel figlio sinistro di u sono minori della chiave contenuta in u;
- Per ogni nodo u, tutte le chiavi contenute nel sottoalbero radicato nel figlio destro di u sono maggiori della chiave contenuta in u.

"***Dicotomia***" significa *dividere in due*. In pratica, ogni volta che viene confrontata una chiave con quella del nodo corrente, si decide se "scendere a sinistra" o "scendere a destra", escludendo metà dell'albero.

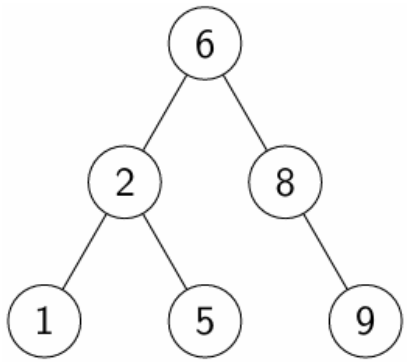
In questo modo tutte le operazioni di ricerca, inserimento e cancellazione sfruttano questa divisione **per ridurre progressivamente lo spazio di ricerca** (agevolare la ricerca).

Alcune precisazioni

Le chiavi dei nodi sono **maggiori** o **minori strette**, perché l'idea generale è che si utilizza una struttura dati per un dizionario, quindi **non è possibile che per una chiave** nella struttura **siano associati più valori**, ad esempio, non posso avere due nodi così strutturati:

- `(10, Alberto)`
- `(10, Roberto)`

Però, nulla vieta che ad una chiave sia associato un insieme (dinamico o non), ad esempio, `(10, ["Alberto", "Roberto"])`



Le operazioni previste dagli alberi binari di ricerca sono:

- **Ricerca** di un elemento nell'insieme;
- **Inserimento** di un elemento nell'insieme;
- Ricerca del **minimo** e del **massimo** dell'insieme;
- **Successore** e **predecessore** di un elemento nell'insieme;
- **Cancellazione** di un elemento dall'insieme.

1.4.1 Implementazione della ricerca

Negli alberi binari di ricerca, la ricerca di un nodo può essere effettuata in modalità **ricorsiva** oppure in modalità **iterativa**.

Ricerca ricorsiva

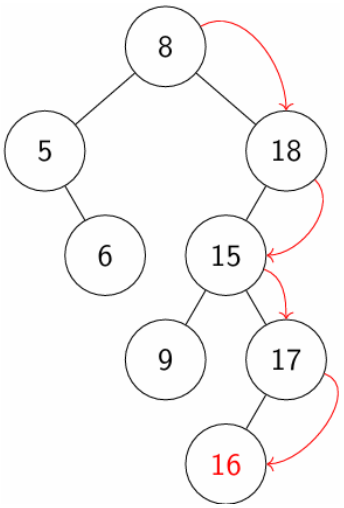
```
1  // ricerca RICORSIVA della chiave x (chiave == data)
2  bstree *search(bstree *p, int x) {
3  //se trovo x oppure ho finito la ricorsione allora ritorno p
4  if(p==NULL || p->data==x) return p;
5
6  // x > data...search the right subtree
7  else if(x > p->data) return search(p->right, x);
8
9  //x < data ... search the left subtree
10 else return search(p->left, x);
11 }
```

- **Tipo di ritorno**: la funzione restituisce un puntatore a un nodo della struttura `bstree`, la cui composizione è identica a quella di un albero binario vista al capitolo 1.2.1.
  Il nodo restituito è:
  - Il **nodo trovato** se la chiave `x` è presente nell'albero;
  - Oppure `NULL`, se la chiave non esiste, cioè **se la ricerca termina su un puntatore nullo**.

  Dunque, il tipo di ritorno consente di sapere **dove si trova l'elemento**, oppure **che non esiste**.

- **Parametri**: la funzione `bstree` accetta due parametri;
  - `bstree *p`: è il puntatore al nodo corrente dell'albero binario di ricerca.
    In genere la prima chiamata della funzione riceve la radice dell'albero (`root`), ma nelle chiamate ricorsive il parametro `p` sarà aggiornato ai figli sinistro o destro.
  - `int x`: rappresenta la chiave da cercare (in questo caso un intero). Si confronta con il campo data contenuto nel nodo corrente.
- **Funzionamento del codice**: la funzione implementa una **ricerca dicotomica ricorsiva**. Nell'esempio riportato in Figure 34, viene effettuata la **ricerca del valore** 16.

Figure 34: Albero BST



  - *riga 4*: la prima istruzione della funzione è un **controllo di terminazione** per fare in modo che, al ripetere della funzione ricorsiva, si capisca se la chiave del nodo è quella cercata o meno; Se `p==NULL`, significa che abbiamo raggiunto un ramo vuoto, la chiave non esiste e la funzione restituisce `NULL`, mentre se `p->data==x` abbiamo trovato il nodo desiderato e si ritorna il puntatore al nodo stesso.
  - *riga 7*: questa seconda istruzione implementa la ricerca ricorsiva nel sotto albero destro. Se la chiave cercata `x` **è maggiore della chiave del nodo corrente** (`p->data`), allora, secondo la proprietà del BST (capitolo 1.4), il **valore** può trovarsi solo nel **sottoalbero destro**.
    Dunque, la funzione richiama se stessa (in modo ricorsivo), passando come parametri il figlio destro del nodo sul quale si stava lavorando (`p->right`), e ancora una volta la chiave cercata (`x`). A questo punto la funzione ricomincia, e se il controllo di terminazione non si attiva, viene ricontrollato se `x` è maggiore o minore di `p->data`.
  - *riga 10*: anche in questo caso viene implementata la ricerca ricorsiva ma nel sottoalbero sinistro. È **importante notare** che non viene specificato `x<p->left` proprio perché come descritto al capitolo capitolo 1.4 ("*Alcune precisazioni*"), **non possono essere presenti nodi con lo stesso valore di chiave**, dunque è scontato che sia l'unica altra opzione possibile.
    Quindi, se la chiave cercata `x` **è minore della chiave del nodo corrente** (`p->data`), allora, secondo la proprietà del BST, il **valore** può trovarsi solo nel **sottoalbero sinistro**.
    Anche in questo caso, viene richiamata ricorsivamente la funzione, con `p->left` come nuovo nodo di partenza, fino a che non si attiva il controllo di terminazione, o `x` è maggiore di `p->data`.

Ricerca iterativa

```
1 // ricerca ITERATIVA della chiave x (chiave == data)
2 bstree *itsearch(bstree *p, int x) {
3 bstree *q = p;
4 while (q && x!= q->data){
5     if (x < q->data) q = q->left;
6     else q = q->right;
7 }
8 return q;
9 }
```

Il **tipo di ritorno** e i **parametri**, sono uguali a quelli della ricerca in modalità ricorsiva, ciò che cambia è la **struttura interna della funzione**, che però produce lo stesso risultato.

- **Funzionamento del codice**: in questo caso si effettua una **ricerca dicotomica**, ovvero ad ogni passo si elimina metà dell'albero possibile, usando però una modalità iterativa.
  - *riga 3*: viene creato un puntatore locale (`*q`) alla struttura `bstree`, il quale viene fatto puntare al parametro `p` che di solito rappresenta la radice (`root`). In questo modo partendo da `p`, `*q` verra usato per scorrere l'albero.
  - *riga 4*: il while continua finché il nodo esiste e la chiave non è stata trovata. L'**uscita dal while** rappresenta il **controllo di terminazione**: se `q` è `NULL` la chiave non esiste, altrimenti è stata trovata.
  - *riga 5-6*: viene definito il controllo condizionale per fare in modo di "scorrere" l'albero a destra se `x > q->data` o a sinistra se `x < q->data`.
  - *riga 8*: quando la condizione del `while` non viene più rispettata, la chiave del nodo viene restituita.

Esempio di utilizzo (valido per entrambi i tipi di ricerca)
Ipotizziamo di voler effettuare la ricerca del nodo 16, come illustrato in Figure 34.

```
int main(int argc, char *argv[]) {
bstree *root = NULL;
root = malloc(sizeof(bstree));

// Allocazione di tutti i nodi e assegnazione di chiave-valore per
// ognuno  di essi, secondo la logica dicotomica (Figure 34).
// ...
// ...

bstree *res = search(root, 16); // Per la ricerca ricorsiva
//bstree *res = itsearch(root, 16); // Per la ricerca iterativa

if(res) printf("chiave trovata! valore = %d\n", res->data);
else printf("chiave NON presente nell'albero\n");

return 0;
}
```

1.4.2 Creazione di un nuovo nodo

```
1 // Funzione per creare un nodo
2 bstree *new_node(int x){
3 bstree *p;
4 p = malloc(sizeof(bstree));
5 p->data = x;
6 p->left = NULL;
7 p->right = NULL;
8 return p;
9 }
```

- **Tipo di ritorno**: la funzione restituisce un puntatore alla struttura del nodo appena creato.
- **Parametri**: la funzione `new_node()` accetta come unico parametro un intero (`int x`), ovvero la chiave da salvare nel nodo.
- **Funzionamento del codice**: vengono definite le istruzioni per assegnare al nuovo nodo il valore della chiave e dei suoi figli sinistro e destro.
  - *riga 3-4*: viene dichiarato il puntatore `p` alla struttura `bstree`. Successivamente, tramite `malloc(sizeof(bstree))` viene allocata dinamicamente la memoria necessaria a contenere un nodo, e il puntatore `p` viene fatto puntare a questa nuova area di memoria.
  - *riga 5*: viene assegnato al campo `data` del nodo il valore `x`, fornito come parametro.
  - *riga 6-7*: i puntatori `left` e `right` vengono inizializzati a `NULL`. Ciò significa che, al momento della creazione, **il nodo non ha figli** ed è **una foglia**.
  - *riga 8*: la funzione restituisce il puntatore al nodo creato.

1.4.3 Inserimento di un nodo

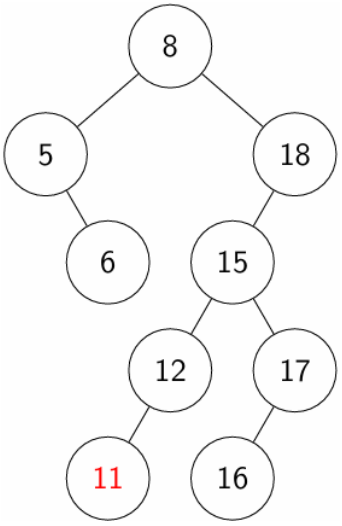
Come per la ricerca dei nodi, anche l'**inserimento di un nodo** all'interno dell'albero può avvenire mediante due modalità: **ricorsiva** o **iterativa**.

Inserimento ricorsivo

```
1  // Funzione per inserire un nodo
2  bstree *insert(bstree *p, int x){
3  //void tree
4  if(p==NULL) return new_node(x);
5
6  //insert to right
7  else if(x > p->data) p->right = insert(p->right, x);
8
9  //insert to left
10 else p->left = insert(p->left, x);
11
12 return p;
13 }
```

- **Tipo di ritorno**: la funzione restituisce un puntatore a `bstree`.
  Il valore restituito rappresenta la radice del sottoalbero risultante dopo l'inserimento;
- **Parametri**: la funzione accetta due parametri;
  - `bstree *p`: è il puntatore al nodo corrente dell'albero binario di ricerca.
    In genere la prima chiamata della funzione riceve la radice dell'albero (`root`), ma nelle chiamate ricorsive il parametro `p` sarà aggiornato ai figli sinistro o destro.
  - `int x`: è la chiave da inserire nel campo `data` del nuovo nodo.
- **Funzionamento del codice**: la struttura del codice è simile a quella che viene utilizzata per effettuare la ricerca di un nodo (capitolo 1.4.1). Come avviene per la ricerca di un nodo, anche l'**inserimento sfrutta la logica dicotomica** utilizzata per la creazione dell'albero binario in modo tale che **in base al valore della chiave del nodo che si vuole inserire**, venga "*esclusa*" una metà dell'albero ad ogni ricorsione o iterazione (in questo caso ad ogni ricorsione).
  Nell'esempio riportato in Figure 35 viene effettuato l'inserimento del valore 11.

Figure 35: Albero BST



- *riga 3*: se il primo controllo condizionale è verificato (`p=NULL`), significa che siamo arrivati alla posizione corretta dove deve essere inserito il nuovo nodo. Una volta trovata la posizione corretta, viene creato e restituito un nuovo nodo tramite la funzione `new_node(x)` vista al capitolo 1.4.2.
- *riga 7*: in questo secondo controllo condizionale viene controllato se la chiave `x` da inserire sia maggiore della chiave del nodo corrente (`x > p->data`). In caso affermativo viene richiamata la funzione in modo ricorsivo sul figlio destro del nodo corrente (`p->right`), come specificato dalle proprietà dei BST.
- *riga 9*: il terzo controllo condizionale viene eseguito nel caso in cui il nodo corrente **non sia nullo** (`p!=NULL`) e `x` **non sia maggiore** di `p->data`. Ciò vuol dire che `x < p->data` e dunque, secondo le proprietà dei BST, la funzione viene richiamata in modo ricorsivo sul figlio sinistro del nodo corrente (`p->left`).
- *riga 11*: la funzione ritorna sempre `p`, cioè la radice del sottoalbero risultante, o per meglio dire, il padre del figlio appena creato.

Inserimento iterativo

```
1  bstree *itinsert(bstree *p, int x){
2  bstree *y = NULL, *q = p;
3  while(q){ //scendo fino ad una foglia
4      y = q;
5      if (q->data > x) q = q->left;
6      else q = q->right;
7  }
8  if(y==NULL) return new_node(x); // se albero vuoto
9  else if(y->data > x)  y->left=new_node(x); // inserisco sx
10 else y->right=new_node(x); // inserisco dx
11
12 return p;
13 }
```

Il **tipo di ritorno** e i **parametri**, sono uguali a quelli dell'inserimento in modalità ricorsiva, ciò che cambia è la **struttura interna della funzione**, che però produce lo stesso risultato.

- **Funzionamento del codice**: il procedimento sfrutta la stessa logica dicotomica vista nella versione ricorsiva, ma la navigazione avviene mediante un ciclo `while`. La visita scende nell'albero "*scorrendo*" ogni volta il puntatore verso il figlio sinistro o destro, finché non si arriva alla posizione corretta per inserire il nuovo nodo.
  - *riga 2*: vengono dichiarati due puntatori alla struttura `bstree`.
    → `*q`: viene fatto puntare a `p`, che nella fase iniziale rappresenta il nodo radice (`root`);
    → `*y`: viene fatto puntare a `NULL`. Il motivo è che in seguito, `y` e `q` verranno fatti scorrere assieme, in questo modo `y` partendo "*in ritardo*" memorizzerà sempre il nodo precedente a `q`. Da questo punto di vista si può dire che `y` sia sempre il padre di `q`.
  - *riga 3*: subito dopo viene utilizzato un ciclo `while` per controllare che il nodo corrente `q` non sia `NULL`. In questo modo si ha la sicurezza che il nodo corrente non sia una foglia ed

è possibile definire le successive istruzioni per poter scorrere l'albero in base alla chiave che si vuole ricercare.
- *riga 4-6*: alla prima iterazione, il nodo `y` che puntava a `NULL` prende il nodo `q`; In questo modo, in base al valore di `x` (controllo condizionale), `q` viene fatto scorrere verso il figlio destro o sinistro e `y` memorizzerà il nodo che è considerato il padre di questi ultimi.

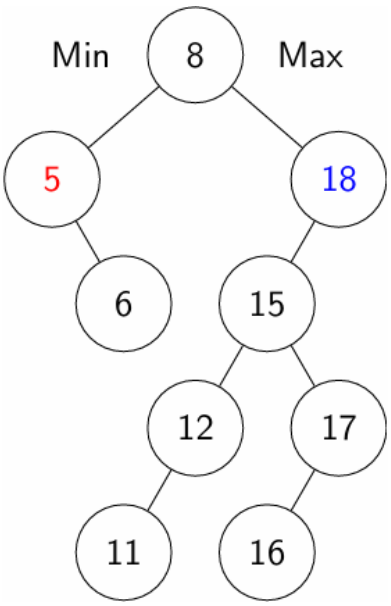
Quando si esce dal ciclo `while` significa che `q`, andando a destra o a sinistra è arrivato ad un nodo `NULL`, quindi oltre una foglia. A questo punto viene eseguito uno dei controlli condizionali:
- *riga 8*: in questo controllo condizionale si gestisce il caso in cui `y` sia rimasto a `NULL`. Quando succede ciò, significa che anche `q` era a `NULL` e dunque l'albero era vuoto: viene quindi richiamata la funzione `new_node(x)` per creare ed inserire il primo nodo dell'albero.
- *riga 9-10*: in questi ultimi due controlli condizionali viene deciso se creare un figlio sinistro o un figlio destro. Si è detto che il puntatore `y` rappresenta il padre di `q`, in altre parole sarà l'ultimo nodo valido (foglia) alla terminazione del ciclo `while`.
  Dunque, per capire se il nuovo nodo debba essere un figlio destro si controlla che `x > y->data`, mentre per capire se debba essere un figlio sinistro si controlla che `x < y->data`.
- *riga 12*: viene restituita la radice originale dell'albero.

Esempio di utilizzo (valido per entrambi i tipi di inserimento)
Ipotizziamo di voler effettuare l'inserimento del nodo 11, come illustrato in Figure 35.

```
int main(int argc, char *argv[]) {
bstree *root = NULL;
/* N.B: In questo caso la malloc non serve perche' viene effettuata
dalla funzione di inserimento quando necessario tramite la funzione
"new_node()" (quindi solitamente al primo passaggio). */

// Inserimento ricorsivo dei valori
root = insert(root, 8);
root = insert(root, 5);
// ...
// ...
root = insert(root, 11);

// Per l'inserimento iterativo basterebbe sostituire:
// root = itinsert(root, 11);
return 0;
}
```

1.4.4 Ricerca del massimo e del minimo

Ancora una volta la logica dicotomica con la quale vengono creati gli alberi binari torna utile per la ricerca del nodo dell'albero con valore di chiave massimo o minimo.
Partendo dalla radice, per trovare il **nodo minimo** va ispezionato sempre il **lato sinistro dell'albero binario**, mentre per trovare il **nodo massimo** va ispezionato **lato destro** dell'albero.



Esempio pratico

- *Nodo minimo*: Per trovare il **nodo con valore di chiave minimo** all'interno dell'albero binario, si effettua una ricerca sempre verso sinistra. Il nodo minimo è quello il cui figlio sinistro è `NULL` (nel caso dell'immagine il nodo 5);
- *Nodo massimo*: Per trovare il **nodo con valore di chiave massimo** all'interno dell'albero binario, si effettua una ricerca sempre verso destra. Il nodo minimo è quello il cui figlio destro è `NULL` (nel caso dell'immagine il nodo 18);

Lo stesso discorso vale per trovare il **massimo o il minimo di un sottoalbero** radicato a partire da un nodo specifico dell'albero: ad esempio, il massimo del sottoalbero radicato nel nodo con valore di chiave 17 è la radice stessa poiché il suo figlio destro è `NULL`.

Anche la ricerca del massimo e del minimo possono essere effettuate in modalità **ricorsiva** o in modalità **iterativa**. Il **costo** di entrambe le operazioni di ricerca **è proporzionale all'altezza dell'albero**.

Ricerca ricorsiva del massimo e del minimo

```
1  // Function to find the maximun value
2  bstree *find_maximum(bstree *p) {
3  if(p == NULL) return NULL;
4  else if(p->right != NULL) return find_maximum(p->right);
5  return p;
6  }
7
8  // Function to find the minimum value
9  bstree *find_minimum(bstree *p) {
10 if(p == NULL) return NULL;
11 else if(p->left != NULL) return find_minimum(p->left);
12 return p;
13 }
```

- **Tipo di ritorno**: la funzioni restituiscono un puntatore a `bstree`. Se l'albero non è vuoto, il valore restituito sarà il nodo che contiene la chiave massima o minima del BST;
- **Parametri**: entrambe le funzioni accettano un solo parametro, ovvero un puntatore alla struttura di un nodo `bstree *p`. In base alla logica della funzione, `p` è il puntatore al nodo dal quale si vuole iniziare la ricerca del massimo o del minimo.
  Solitamente si utilizza `root` se si vuole esaminare l'intero albero, ma è possibile passare anche il puntatore alla radice di un sottoalbero.
- **Funzionamento del codice**: come detto precedentemente, la logica dietro la ricerca del valore massimo e del valore minimo è abbastanza semplice.
  - *riga 3/10*: se il puntatore passato come parametro è nullo (`p==NULL`), significa che l'albero è vuoto e non è possibile determinare alcun valore massimo o minimo;
  - *riga 4/11*: per implementare ciò che è stato detto precedentemente viene effettuato un controllo condizionale che differisce in base al tipo di funzione:

→ nel caso della ricerca del massimo si verifica se il **figlio destro** del nodo corrente non è nullo;
→ nel caso della ricerca del minimo si verifica se il **figlio sinistro** del nodo corrente non è nullo.

Nel caso in cui il **controllo condizionale sia vero**, la funzione viene richiamata in modo ricorsivo specificando il figlio verso il quale ci si vuole spostare (`p->right` per il massimo e `p->left` per il minimo), in modo tale che la funzione ricominci con la posizione successiva da analizzare.

- *riga 5/12*: se invece i controlli condizionali risultassero falsi, significherebbe che il massimo/minimo è stato trovato perché il figlio destro/sinistro non esiste (`p->right==NULL` o `p->left==NULL`), uscendo così dalla ricorsione.
  A questo punto viene restituito `p`, ovvero l'ultimo nodo valido visitato, il quale rappresenterà il valore massimo/minimo dell'albero.

Ricerca iterativa del massimo e del minimo

```
1  bstree *itfind_minimum(bstree *p) {
2  if(p == NULL) return NULL;
3  while (p->left != NULL) p = p->left;
4  return p;
5  }
6
7  bstree *itfind_maximum(bstree *p) {
8  if(p == NULL) return NULL;
9  while (p->right != NULL) p = p->right;
10 return p;
11 }
```

Il **tipo di ritorno** e i **parametri**, sono uguali a quelli che vengono utilizzati con la modalità ricorsiva, cio che cambia è la struttura interna della funzione che però produce lo stesso risultato.
Utilizzando un'altra modalità di esecuzione, il **funzionamento del codice** per forza di cose cambia, ma la funzione rimane comunque **molto semplice** e la maggior parte delle istruzioni rimangono invariate. I controlli condizionali che nella modalità ricorsiva vengono utilizzati per capire quando richiamare ricorsivamente la funzione, in questo caso vengono utilizzati come guardia nel ciclo `while` per fare in modo di "scorrere" l'albero a destra o a sinistra in base al valore che si vuole trovare (massimo o minimo).

Esempio di utilizzo (valido per entrambe le modalità)

```
1  int main(int argc, char *argv[]) {
2  bstree *root = NULL;
3  /* Creazione di un albero BST in logica dicotomica utilizzando la
4  funzione di inserimento, come fatto al capitolo (1.4.3) */
5  // ...
6  // ...
7
8  // Ricerca ricorsiva del massimo e del minimo
9  bstree *min_node = find_minimum(root);
10 bstree *max_node = find_maximum(root);
```

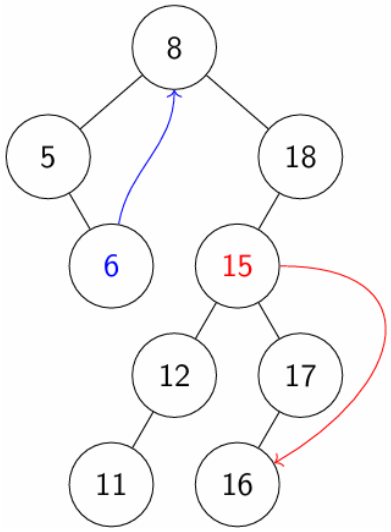
```c

// Ricerca iterativa del massimo e del minimo
// bstree *min_node = itfind_minimum(root);
// bstree *max_node = itfind_maximum(root);

printf("Minimo trovato = %d\n", min_node->data);
printf("Massimo trovato = %d\n", max_node->data);

return 0;
}
```

1.4.5 Ricerca del successore-predecessore

Definizione di successore

Il **successore** di un nodo u è il più piccolo nodo maggiore di u.
Allo stesso modo, possiamo dire che il **predecessore** di un nodo u è il più grande nodo minore di u.

Figure 37: Albero BST



Per trovare il successore di u, sono dati due casi:

- *Caso 1*: Se il nodo u **ha un figlio destro**, il successore v è il minimo nodo del sottoalbero destro; Dunque in questo caso la situazione è molto semplice perché basta utilizzare la funzione vista al capitolo 1.4.4 per la ricerca del minimo.
  Trovando il nodo più piccolo, una **caratteristica determinante** di questo caso è che il successore non ha figlio sinistro.
- *Caso 2*: Se il nodo u **non ha un figlio destro**, il successore è il **primo antenato** (v) di u, per cui u sta nel sottoalbero sinistro dell'antenato v.
  Ad esempio, ipotizziamo di voler cercare il successore del nodo 6 come illustrato in Figure 37. Per prima cosa si controlla il sottoalbero radicato in 5, e ci si chiede "*u (6) sta nel sottoalbero sinistro di v (5)?*" In questo caso no, infatti 5 non è il successore. Subito dopo, continuando a salire, si arriva al sottoalbero radicato in 8: "*u (6) sta nel sottoalbero sinistro di v (8)?*" In questo caso la risposta è si, dunque 8 è il successore. Ciò è dato anche dal fatto che trovandosi nell'albero sinistro, il primo antenato v che incontro, **per le proprietà dei BST**, sarà sicuramente più grande del nodo u: in questo caso andrà effettuato **un controllo tra le chiavi dei nodi**.

Dunque, la struttura di un albero binario di ricerca consente di determinare il successore di un nodo **senza mai effettuare il confronto tra le chiavi** (quando il successore viene trovato con il *caso 1*) come invece avviene per la ricerca del massimo o del minimo.

Ricerca iterativa del successore

```c
/* Definisco la funzione che ricerca il nodo antenato per gestire il
caso 2 */
bstree* ancestor(bstree *p, bstree *px){
int x = px->data;
```

```
bstree *y = NULL; //l'antenato
bstree *q = p;
while (q && x != q->data){
    y = q;
    if (x < q->data) q = q->left;
    else q = q->right;
}
return y;
}
```

```
// Funzione principale per la ricerca del successore
bstree* successor(bstree *p, bstree *px){
if(px->right) return find_minimum(px->right); //Caso 1
bstree *y = ancestor(p, px); // Caso 2 (discesa)
while (px && y && px == y->right){ //Caso 2 (risalita)
    px = y;
    y = ancestor(p, px);
}
return y;
}
```

Per seguire un filo logico durante il discorso, analizziamo per prima la funzione `successor`.

- **Tipo di ritorno**: la funzione restituisce un puntatore (`y`) al nodo del successore cercato;
- **Parametri**: la funzione accetta in ingresso due parametri;
  - `bstree *p`: un puntatore alla struttura di un nodo generico che viene utilizzato per passare il puntatore alla radice dell'intero albero;
  - `bstree *px`: anche `px` è un puntatore alla struttura di un nodo generico, ma viene utilizzato per passare il puntatore al nodo dell'albero del quale si vuole cercare il successore;
- **Funzionamento del codice**: come detto precedentemente, quando ci si occupa della ricerca del successore, bisogna gestire due casi;
  - *riga 3*: in questa riga viene gestito il caso più semplice (caso 1).
    Viene effettuato un controllo condizionale sull'esistenza del figlio destro sul nodo del quale ci interessa trovare il successore (`px->right`). Se il controllo risulta vero, significa che il nodo `px` avrà un sottoalbero che, per le caratteristiche dei BST, conterrà solo nodi con un valore di chiave maggiore al suo.
    A questo punto, per trovare il successore di `px` basta utilizzare `find_minimum` o `itfind_minimum` sul figlio destro di `px` per trovare il nodo minimo dell'intero sottoalbero;
  - *riga 4*: da questo punto in poi viene gestito il caso in cui il nodo di cui si vuole conoscere il successore **non abbia un figlio destro** (caso 2).

    Funzionamento di `ancestor`: come detto precedentemente, bisogna risalire lungo l'albero, partendo dal nodo di cui vogliamo conoscere il successore. Poiché i nodi non hanno un puntatore al padre, per trovare il primo nodo "sopra" `px` dobbiamo necessariamente scendere partendo dalla radice. La funzione `ancestor` si occupa proprio di trovare il nodo che sta immediatamente sopra `px` lungo il cammino dalla radice. Il **tipo di ritorno** e i **parametri** sono quindi gli stessi della funzione `successor`.
    → *riga 4*: il valore della chiave del nodo `px` viene salvato nella variabile `x`;
    → *riga 5*: viene dichiarato un puntatore `y` e inizializzato a `NULL`: questo punterà di volta

in volta al nodo più recente incontrato nel cammino;

→ *riga 6*: viene creato un puntatore `q` e gli viene assegnata la radice `p`, per poter scorrere tutto l'albero partendo dall'alto;

→ *riga 7*: Per poter effettuare lo scorrimento si utilizza un ciclo `while` la cui condizione è quella di continuare la sua iterazione fino a che il nodo `q` è valido e fino a che le chiavi del nodo `px` e del nodo `q` non coincidono.

→ *riga 8*: ad ogni iterazione il nodo puntato da `q` viene salvato in `y`. In questo modo, **quando finalmente arriva a `px`**, il nodo `y` sarà già impostato al nodo che lo precede lungo il cammino;

→ *riga 9-10*: infine, in queste righe vengono gestiti i controlli condizionali per quanto riguarda lo spostamento all'interno dell'albero e, come sempre fatto per la logica costruttiva degli alberi BST, ci si sposta sul figlio destro se la chiave cercata (x) del nodo `px` è più grande della chiave del nodo attuale `q`, altrimenti ci si sposta sul nodo sinistro.

→ *riga 12*: quando la condizione nel ciclo `while` non è più rispettata si restituice `y`.

Alla fine della funzione `ancestor()`, **`y` contiene il nodo padre (inteso come nodo immediatamente sopra) del nodo `px` lungo il cammino dalla radice.**

- *riga 5*: da qui in poi, nel codice del successore, viene eseguito un ulteriore `while` per verificare se il nodo antenato trovato con `ancestor` sia realmente il **successore** di `px`, oppure se sia necessario risalire ulteriormente.
  Perché risalire? Perché se `px` è figlio destro di `y`, `y` non può essere il suo successore (per le proprietà dei BST).
- *riga 6-7*: a questo punto del codice, una volta entrati nel `while` di risalita, abbiamo appurato che il nodo "appena sopra" il nodo `px` (ovvero (`y`)) non è il suo successore. Ciò vuol dire che non serve più memorizzare il nodo `px` originale, ma bisogna **risalire per trovare il primo nodo che è abbia un figlio sinistro**: il nodo trovato sarà di sicuro il successore del nodo `px` originale poiché avendo un figlio sinistro avrà un valore di chiave maggiore. Per fare ciò all'interno del `while`, `y` diventa il nuovo nodo originale "fittizio" e vine richiamata su di esso la funzione `ancestor()`;
- *riga 9*: quando la guardia del `while` non è più rispettata siamo sicuri di aver trovato il successore del nodo originale restituendo `y` che contiene il risultato di `ancestors()`.

Esempio di utilizzo

Ipotizziamo di voler effettuare la ricerca del successore del nodo 17 come in Figure 37.

```
int main(int argc, char *argv[]) {
bstree *root = NULL;
/* Creazione dell'albero BST in logica dicotomica illustrato in
Figure 37 utilizzando la funzione di inserimento, come fatto al
capitolo (1.4.3) */
// ...
// ...

bstree *px = itsearch(root, 17); // Scelgo un nodo
bstree *succ = successor(root, px); // Individuo il suo successore
printf("Il successore del nodo %d e' %d\n", px->data, succ->data);
}
```

Passaggio da successore a predecessore

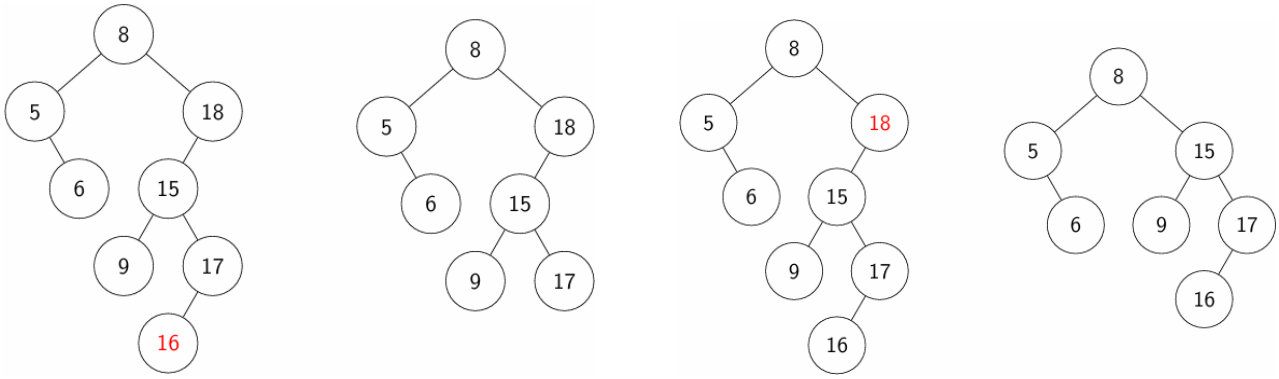
Il discorso intrapreso per quando riguarda la ricerca del successore ovviamente vale anche per la ricerca del **predecessore** di un determinato nod all'interno dell albero.

- Il ragionamento con il *caso 1* e il *caso 2* vengono effettuati tenendo in considerazione il **figlio sinistro** e **non** il figlio destro;
- Nello specifico, per il *caso 1* al posto della funzione `find_minimum()` si utilizzerà `find_maximum()`.

1.4.6 Cancellazione di un nodo

Tramite la cancellazione di un nodo viene **rimossa la chiave** k **dall'albero** T.
Durante la cancellazione di un determinato nodo possono sorgere **tre differenti casi**:

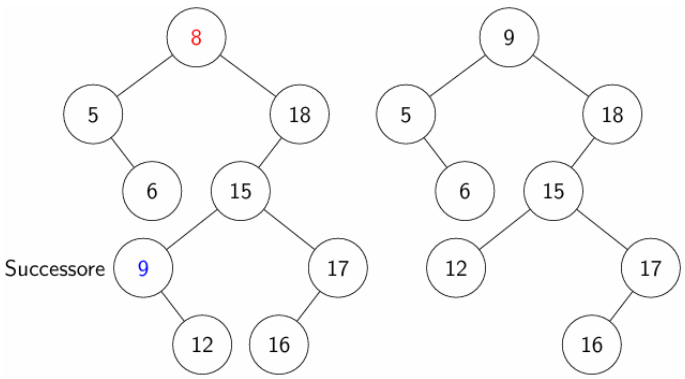
- *Caso 1* - ***Il nodo*** u ***da eliminare non ha figli***: in questo caso u è una foglia e viene semplicemente rimossa. **Eliminare le foglie non cambia l'ordine dei nodi rimanenti**;
- *Caso 2* - ***Il nodo*** u ***da eliminare ha un solo figlio*** f **(destro o sinistro)**: in questo caso si elimina il nodo u rendendo f figlio del padre di u.
  Come si può notare nell'esempio *(b)*, nonostante il nodo con valore di chiave 18 sia la radice di un sottoalbero, quest'ultimo può essere collegato senza alcun tipo di problema al nodo con valore di chiave 8. Infatti, 18 era il figlio destro del nodo 8, dunque tutti i nodi sotto di esso saranno comunque maggiori di 8, **rispettando così le proprietà degli alberi BST**.



(a) *Caso 1 - nessun figlio* (b) *Caso 2 - un solo figlio*

- *Caso 3* - ***Il nodo*** u ***da eliminare ha due figli***: in questo caso la logica dietro l'eliminazione del nodo diventa leggermente più complessa. Come illustrato in Figure 38, supponiamo di voler eliminare il nodo u con valore di chiave 8. Per prima cosa bisogna individuare il successore di u, ovvero, come detto al capitolo 1.4.5, "*il più piccolo valore più grande di* u": quando si va a rimuovere un nodo con due figli, il suo **successore** ha la caratteristica di essere il sostituto adatto a ricoprire quella posizione in modo tale che le proprietà degli alberi BST vengano rispettate.

Figure 38: Cancellazione di un nodo



A questo punto possiamo dire che il successore (v) **rispetti le seguenti proprietà**:

- È sicuramente **maggiore** dei nodi nel sottoalbero sinistro di u;
- È sicuramente **minore** dei nodi nel sottoalbero destro di u.

Una volta che il successore è stato individuato, bisogna sovrascriverlo al nodo che si vuole cancellare, eliminando anch'esso dalla sua posizione attuale.

A questo punto il *caso 3* si può **ridurre ad uno dei due casi** visti in precedenza:

- *Caso 1*: **il successore è una foglia**, quindi può essere sovrascritto e rimosso senza ulteriori accorgimenti;
- *Caso 2*: **il successore ha solo un figlio destro**.
  A differenza di un normale *caso 2*, quando si arriva ad esso passando dal *caso 3*, **non può mai capitare** che il **successore** abbia **solo figlio sinistro**, perché se avesse il figlio sinistro **non sarebbe il minimo** del sottoalbero destro (capitolo 1.4.4 - caso 1).

Cancellazione ricorsiva di un nodo

```
1  bstree *deleteNode(bstree *p, int x) {
2  if(p==NULL) return NULL;
3  if (x > p->data) p->right = deleteNode(p->right, x);
4  else if(x < p->data) p->left = deleteNode(p->left, x);
5  else{ //node found is p
6      if(p->left==NULL && p->right==NULL){ //Nessun figlo (caso 1)
7          free(p);
8          return NULL;
9      }
10     else if(p->left==NULL || p->right==NULL){ //Un figlio (caso 2)
11         bstree *temp;
12         if(p->left==NULL) temp = p->right;
13         else temp = p->left;
14         free(p);
15         return temp;
16     }
17     else{ //Due figli (caso 3)
18         bstree *temp = find_minimum(p->right);
19         p->data = temp->data;
20         p->right = deleteNode(p->right, temp->data);
21     }
22 }
23 return p;
24 }
```

- **Tipo di ritorno**: la procedura di cancellazione di un nodo restituisce un puntatore alla radice dell'albero, che **può cambiare** se il nodo cancellato **è proprio la radice**;
- **Parametri**: la funzione accetta in ingresso due parametri;
  - `bstree *p`: un puntatore alla struttura di un nodo generico che viene utilizzato per passare il puntatore alla radice dell'intero albero;
  - `int x`: un valore intero `x`, che rappresenta la chiave del nodo da cancellare.
- **Funzionamento del codice**: All'interno della funzione `deleteNode()` vengono gestiti tutti i casi di cancellazione del nodo.
  - *riga 2*: si gestisce il caso base il caso base: se il puntatore `p` è `NULL`, vuol dire che non esiste alcun albero o che non è stato trovato alcun nodo con valore `x`. In questo caso viene semplicemente restituito `NULL`;

- *riga 3-4*: in queste due righe vengono gestiti i controlli codizionali per la ricera del nodo da rimuovere all'interno dell'albero. Se la chiave `x` è maggiore della chiave del nodo corrente (`x > p->data`), allora la ricerca avverrà richiamando ricorsivamente `deleteNode()` sulla radice del sottoalbero destro, altrimenti (`x < p->data`) la ricerca avviene, sempre in modo ricorsivo, ma sul sottoalbero sinistro;
- *riga 5*: se si entra in quest'ultimo caso significa che il nodo da eliminare è stato trovato (`x == p->data`). All'interno di esso vengono implementati i 3 casi differenti per gestire l'eliminazione del nodo in base alla sua posizione all'interno dell'albero;
- *riga 6-8*: il primo controllo condizionale gestisce il caso in cui il nodo da rimuovere sia una foglia (*caso 1*). Quest'ultimo è valido solo se entrambi i puntatori ai figli destro e sinistro sono `NULL`. Verificato ciò, si procede ad eliminare il nodo `p` tramite il comando `free()` e viene ritornato `NULL` come nuovo puntatore da assegnare al padre (o alla radice se `p` era la radice).;
- *riga 10-16*: il secondo controllo condizionale gestisce il caso in cui il nodo da rimuovere abbia un solo figlio (che sia destro o sinistro, ovvero *caso 2*). Quest'ultimo è valido solo se almeno uno dei due puntatori ai figli è nullo. Una volta all'interno del controllo condizionale è bene **prestare attenzione anche al meccanismo di ricorsione** che viene utilizzato per permettere di **salvare il figlio del nodo** che si vuole eliminare.

  Facendo riferimento alla Figure 38, ipotizziamo di voler rimuovere il nodo con valore di chiave 9 ed essere posizionati in 15. La funzione inizia e ci si ferma nella *riga 4* (poiché $9 < 15$) dove avvengono due cose:
  $\rightarrow$ Si scende a sinistra perché viene richiamata `deleteNode()` su `p->left`, ovvero 9;
  $\rightarrow$ Allo stesso tempo, il puntatore al figlio sinistro (9) prenderà il risultato della chiamata appena avvenuta su `deleteNode()`.
  Una volta su 9, si entra nell'ultimo `else` e, in seguito, nel controllo condizionale per i nodi con un solo figlio, poiché l'unico figlio di 9 è 12. Viene dichiarato un puntatore generico alla struttura di un nodo `temp`: quest'ultimo verrà utilizzato per salvare temporaneamente il figlio (12), così da poter fare la `free()` del nodo 9 senza perderlo.
  A questo punto `temp` viene restituito a `p->left` (il puntatore al figlio sinistro di 15) come risultato della funzione ricorsiva chiamata prima: in questo modo, 12 diventa figlio sinistro diretto di 15.
- *riga 18-21*: il terzo controllo condizionale gestisce il caso in cui il nodo da rimuovere abbia figlio destro e sinistro, ed ovviamente è valido se non si entra negli altri due casi prima. Precedentemente si è detto che, in teoria, per eliminare un nodo con due figli si dovrebbe trovare il suo successore (capitolo 1.4.5). Dal punto di vista implementativo, però, nel BST il successore coincide sempre con il nodo avente il valore minimo nel sottoalbero destro. Per questo motivo, nel codice non viene richiamata esplicitamente una funzione `successor()`, ma si utilizza direttamente la funzione `find_minimum(p->right)` (capitolo 1.4.4), che rappresenta esattamente tale concetto.

  Trovato il suo successore, il valore della chiave di quest'ultimo viene semplicemente sovrascritto (`p->data = temp->data`) sul nodo da eliminare.
  A questo punto il problema è che il successore **esiste ancora fisicamente al suo posto originario**. La sua rimozione è semplice: per definizione il successore di un nodo non ha un figlio sinistro, quindi si richiama ricorsivamente la funzione `deleteNode()` che ne gestirà l'eliminazione seguendo il *caso 1* se è una foglia o il *caso 2* se ha un figlio destro.

Esempio di utilizzo
Ipotizziamo di voler cancellare il nodo 8 come illustrato in Figure 38.

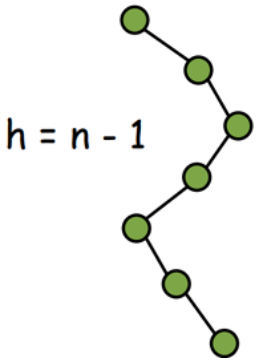
```
int main(int argc, char *argv[]) {
bstree *root = NULL;
/* Creazione dell'albero BST in logica dicotomica illustrato in
Figure 38 utilizzando la funzione di inserimento, come fatto al
capitolo (1.4.3) */
// ...
// ...

root = deleteNode(root, 8); // Cancellazione del nodo 8
printf("La nuova radice e' %d\n", root->data);
}
```

1.4.7 Alberi BST: alcune osservazioni

Tutte le operazioni effettuate fino ad ora (ricerca, inserimento, ecc...) sono confinate ai nodi posizionati lungo ad un cammino semplice dalla radice alla foglia.
Durante tutte queste operazioni un fattore impattante è il **tempo di ricerca**.

Tempo di ricerca

Il **tempo di ricerca** è limitato superiormente da h, dove h è l'altezza dell'albero.



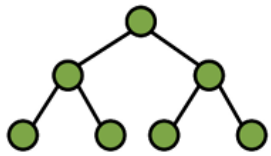
Come si può vedere dall'immagine, avendo un BST di altezza h, il caso pessimo è $O(h)$ dove è previsto di arrivare nella **parte più profonda dell'albero**. Il problema sorge quando si hanno n nodi: il **caso pessimo** della struttura dell'albero è quello che viene generato quando gli **elementi** vengono **inseriti in ordine** perché l'albero può "allungarsi". Ad esempio, si pensi ad un albero dove vengono inseriti i seguenti nodi: `1->2->3->4->5->` e così via ..., ovviamente tutti nel ramo destro. Quindi un albero contenente n nodi, avrà altezza $h = O(n)$ e **tempo di ricerca** $O(h)$. In questo caso si parla di **albero non bilanciato**.
Gli alberi binari di ricerca, sono sì una buona idea per portare la ricerca binaria nel campo degli alberi, però **se i dati sono inseriti in certi modi sbagliati** (caso pessimo), si ottiene un'altezza dell'albero di $O(n)$ e quindi **non si ha nessun vantaggio** rispetto ad una banale **ricerca lineare** (liste).

Quando sono stati sviluppati gli alberi binari di ricerca, si è cercato di capire cosa succede in media, oltre che nel caso pessimo, e si è visto che facendo degli **inserimenti in ordine casuale**, l'altezza media dell'albero è $O(log\ n)$. Nella realtà però, non ci si affida al caso ma si utilizzano delle **tecniche per mantenere l'albero bilanciato**, o per meglio dire, **ribilanciarlo**.

Cos'è un BST bilaciato?

Un BST *"bilanciato"* è un albero binario in cui l'altezza è proporzionale al logaritmo del numero di nodi ($h = log_2(n+1) - 1$).

$h = \log_2 (n + 1) - 1$



Quindi, se l'albero è **ben bilanciato** l'altezza è limitata superiormente esattamente da $h = O(log\ n)$. Quello che si vuole ottenere è **un meccanismo per ribilanciare gli alberi non bilanciati** e fare quindi in modo che la loro altezza sia più simile a $log\ n$ piuttosto che n evitando casi particolari come il precedente. Un approccio possibile è quello degli **alberi AVL**.

1.4.8 Alberi BST: Alberi Adelson-Velsky and Landis (AVL)

Gli alberi AVL introducono alcune proprietà addizionali sugli alberi BST **per fare in modo che rimangano bilanciati**. Una di queste è il **fattore di bilanciamento**.

Fattore di bilanciamento

Il **fattore di bilanciamento** $\beta(v)$ di un nodo v è la differenza di altezza fra i sottoalberi destro e sinistro di v.

Cos'è un albero AVL?

Un albero binario di ricerca è un **albero AVL** se per ogni nodo v l'altezza del sottoalbero sinistro di v e quella del sottoalbero destro di v differiscono al massimo di 1 ($\beta(v) = h(left(v)) - h(right(v))$). In altre parole il **fattore di bilanciamento** deve essere un valore compreso tra $-1 \leqslant \beta(v) \leqslant 1$.

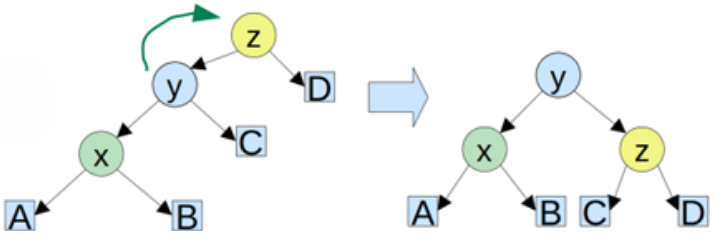
In un albero BST l'inserimento di una nuova foglia in un determinato punto dell'albero può causare uno **sbilanciamento**. Nello specifico il **nodo sbilanciato** è il primo antenato che, dopo l'inserimento del nuovo nodo, presenta un fattore di bilanciamento > 1.
Per evitare gli sbilanciamenti viene utilizzato il concetto di **rotazione**.

Cos'è una rotazione?

Una rotazione è un operazione locale che viene **eseguita** sul **nodo sbilanciato** causando lo spostamento di tre nodi: il nodo sbilanciato stesso, suo figlio e il suo nipote.
Lo sbilanciamento viene riequilibrato **senza violare** le **proprietà strutturali** dei BST.
Dunque, le **rotazioni** permettono di **abbassare** il fattore di bilanciamento.

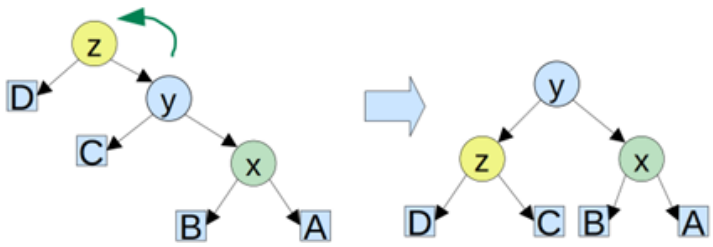
Esistono solo due rotazioni elementari: **rotazione a sinistra** e **rotazone a destra**.
Invece, le **forme di sbilanciamento** di un albero BST sono quattro, e come detto prima, variano in base al punto di inserimento del nuovo nodo:

- *Caso left-left (LL)*: il caso ***left-left*** si verifica quando il nuovo nodo viene inserito nel sottoalbero sinistro del figlio sinistro del primo nodo, il quale diventa sbilanciato.



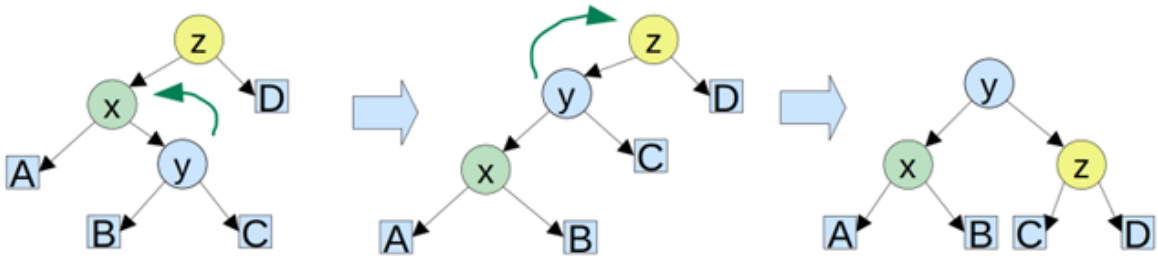
In questo caso il ramo sinistro "pesa" troppo e presenta un **fattore di bilanciamento** > 1. Il caso *left-left* possiede una **forma lineare**, dunque, per ribilanciare l'albero basta effettuare una sola rotazione verso destra.

- *Caso right-right (RR)*: il caso ***right-right*** si verifica quando il nuovo nodo viene inserito nel sottoalbero destro del figlio destro del primo, il quale diventa sbilanciato



Anche in questo caso il ramo destro "pesa" troppo e presenta un **fattore di bilanciamento** < -1. Il caso *right-right* possiede una **forma lineare**, e per ribilanciare l'albero, basta effettuare una sola rotazione verso sinistra.

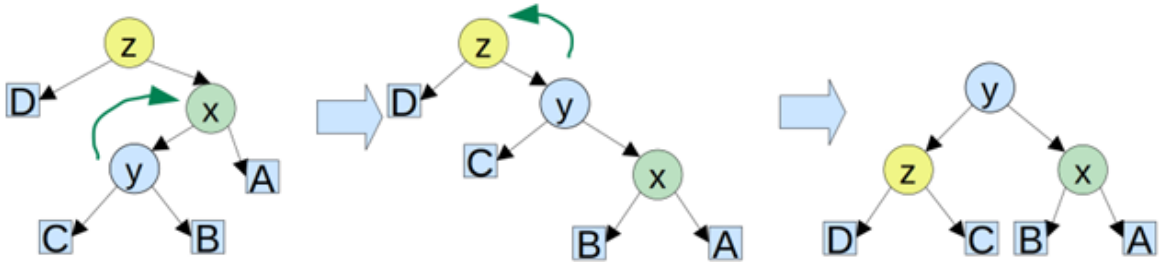
- *Caso left-right (LR)*: il caso ***left-right*** si verifica quando il nuovo nodo viene inserito nel sottoalbero destro del figlio sinistro del primo nodo, il quale diventa sbilanciato.



Il fattore di bilanciamento in questo caso è > 1. In questo caso il sottoalbero presenta una **forma a "zig-zag"** che, per essere bilanciata, necessita di una **doppia rotazione**:

1. **Rotazione a sinistra**: viene effettuata sul figlio destro del figlio sinistro del nodo sbilanciato, per fare in modo di riportarsi alla **forma *left-left***;
2. **Rotazione a destra**: viene effettuata sul nodo sbilanciato per completare il bilancianento.

- *Caso right-left (RL)*: il caso *right-left* si verifica quando il nuovo nodo viene inserito nel sottoalbero sinistro del figlio destro del primo nodo, il quale diventa sbilanciato.



Presenta fattore di bilanciamento < -1, e una forma a "zig-zag" che necessita di una doppia rotazione, in questo caso in senso inverso rispetto alla precedente:

1. **Rotazione a destra**: viene effettuata sul figlio sinistro del figlio destro del nodo sbilanciato, per fare in modo di riportarsi alla **forma *right-right***;
2. **Rotazione a sinistra**: viene effettuata sul nodo sbilanciato per completare il bilancianento.

Rotazione a sinistra/destra per il bilanciamento

```
bstree *rotateleft(bstree *x) {
bstree *y;
y = x->right;
x->right = y->left;
y->left = x;
return (y);
}
```

```
bstree * rotateright(bstree *x) {
bstree *y;
y = x->left;
x->left = y->right;
y->right = x;
return (y);
}
```

- **Tipo di ritorno**: entrambe le funzioni ritornano un puntatore alla struttura di un nodo. Infatti, dopo la rotazione la radice dell'albero cambia: ciò che viene restituito è proprio la nuova radice dell'albero;
- **Parametri**: entrambe le funzioni accettano un unico parametro, ovvero il puntatore alla struttura del nodo che si vuole ruotare (`bstree *x`);
- **Funzionamento del codice**: la logica utilizzata per la creazione delle due funzioni di rotazione è la stessa, ciò che le differenzia è il verso della rotazione (sinistra o destra).

Funzionamento di `rotateleft` (e `rotateright`)
Per illustrare il funzionamento di `rotateleft` utilizziamo la seguente immagine.
Nel caso iniziale ciò che crea lo sbilanciamento è l'aggiunta del nodo C. Dunque deve essere effettuata una rotazione a sinistra specificando il puntatore al nodo A nei parametri della funzione. Ovviamente il funzionamento di `rotateright` è il medesimo, solo invertito.