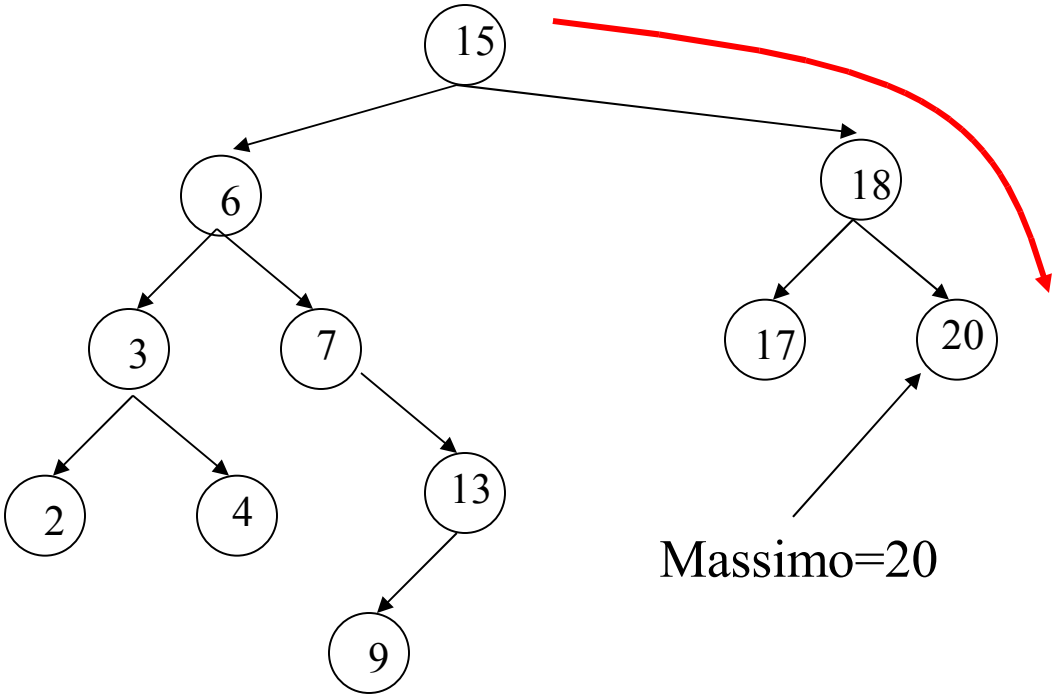
Alberi Binari di Ricerca

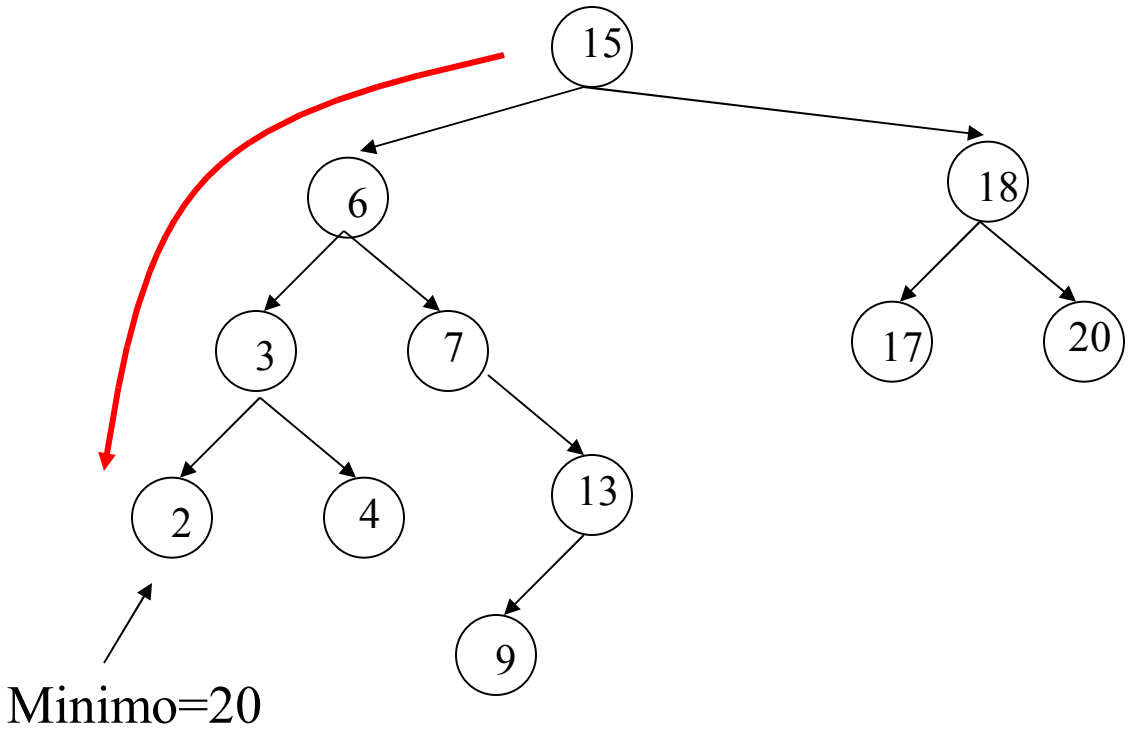
Determinazione della chiave massima e minima

- La chiave massima in un albero binario dovrà trovarsi nel sottoalbero destro della radice
- e nel sottoalbero destro del figlio destro della radice
- e così via
- Analogamente per la chiave minima che dovrà essere nel sottoalbero sinistro
- Pertanto per determinare l'elemento massimo è sufficiente discendere tutti i nodi da figlio destro in figlio destro fino ad arrivare alla foglia (e analogamente con i figli sinistri per il minimo)

Massimo



Minimo



Minimo e massimo

```
Tree-Minimum(x)
1  while left[x] ≠ NIL
2  do    x ← left[x]
3  return x

Tree-Maximum(x)
1  while right[x] ≠ NIL
2  do    x ← right[x]
3  return x
```

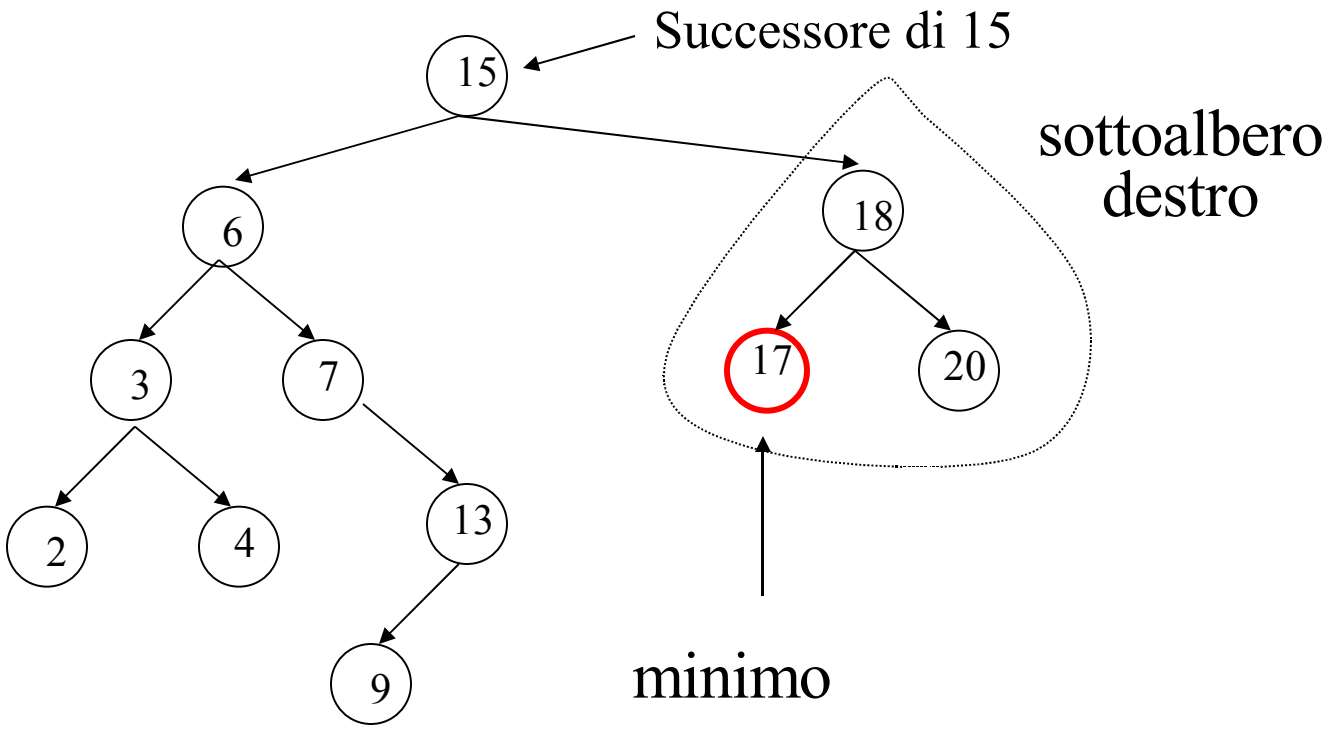
Successore e predecessore

- Dato un nodo nell'albero di ricerca talvolta si richiede di determinare il suo successore (o predecessore) secondo l'ordinamento fornito dalle chiavi.
- Se tutte le chiavi sono distinte, il successore di un nodo x è il nodo con la più piccola chiave maggiore della chiave di x
- Con gli alberi binari di ricerca è possibile determinare il successore (predecessore) di un nodo senza dover confrontare le chiavi

Successore: idea intuitiva

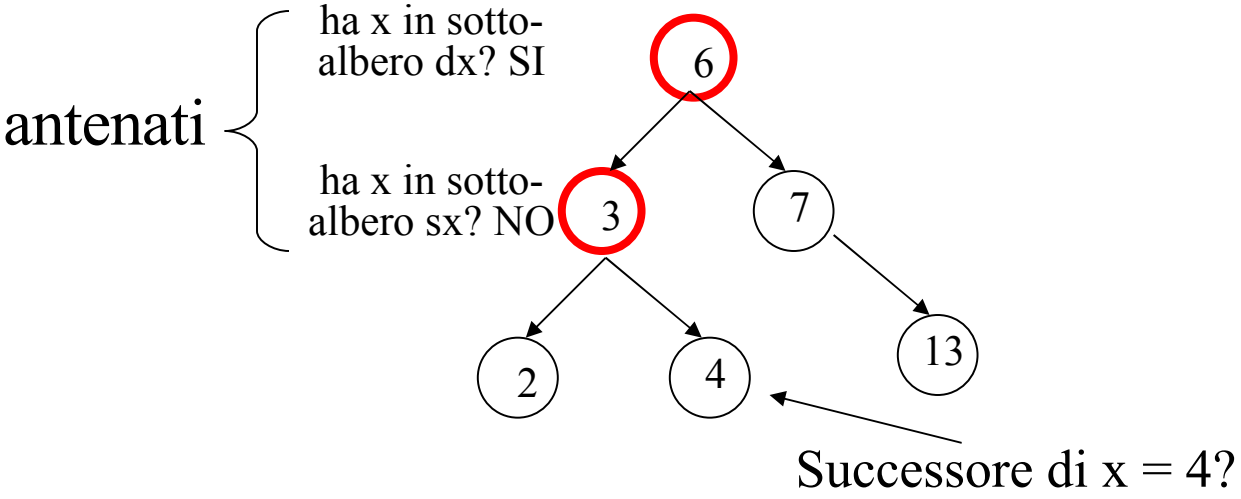
- Si considerano due casi:
  - il nodo x ha un figlio destro
  - il nodo x non ha un figlio destro
- Nel primo caso:
  - si considera il sottoalbero destro che contiene sicuramente nodi con chiavi maggiori della chiave di x
  - in questo sottoalbero il nodo con la chiave più piccola è la foglia alla estrema sinistra, cioè il nodo restituito dalla procedura Tree-Minimum

Successore di nodo con figlio dx



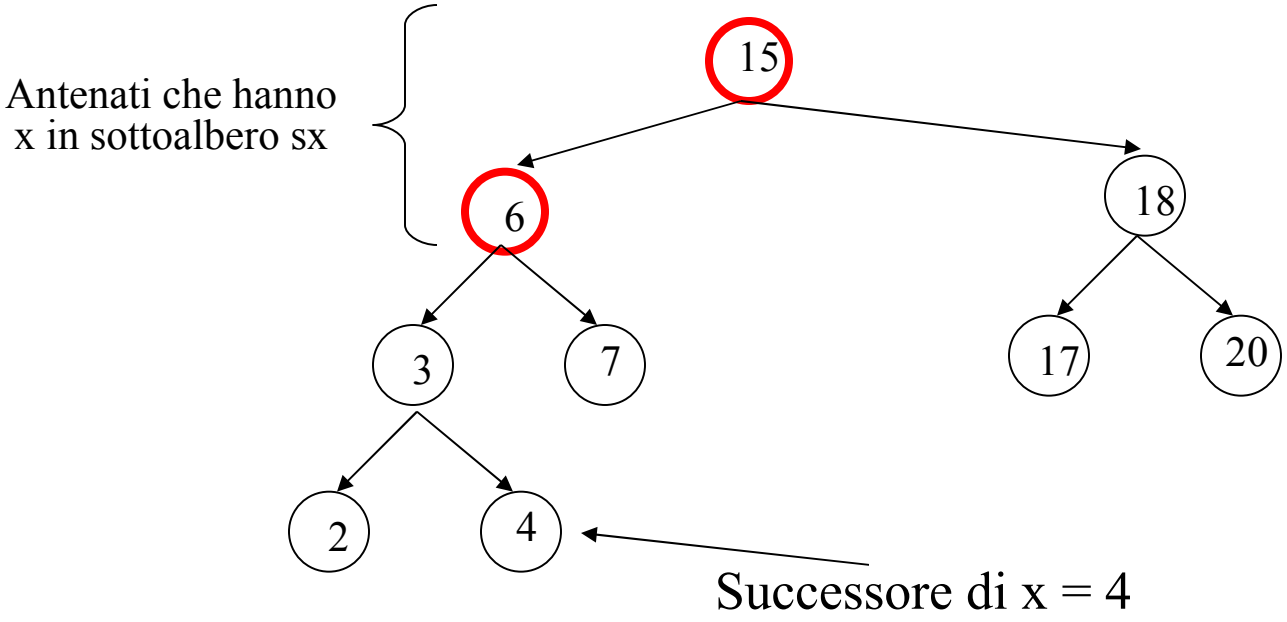
Successore di nodo senza figlio dx

- Nel caso in cui x non ha un figlio destro allora il predecessore deve essere un antenato p di x
- dal punto di vista di p, x deve essere un discendente appartenente ad un sottoalbero sinistro
- infatti le chiavi nel sottoalbero sx hanno chiave minore



Successore di nodo senza figlio dx

- perché la chiave di p sia la più piccola possibile allora p deve essere l'antenato più prossimo
- altrimenti se consideriamo un antenato lontano è vero che ha chiave maggiore, ma..
- esisterà un antenato meno lontano e più piccolo!

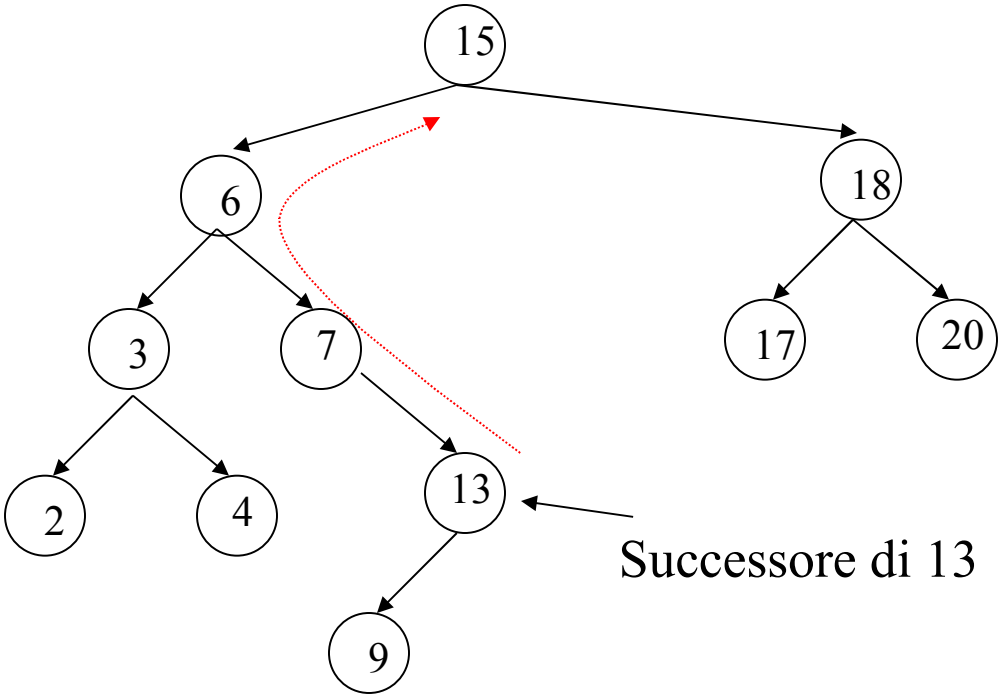


Idea intuitiva

- Per determinare questo nodo antenato è sufficiente risalire gli antenati di x fino a quando non si trova un nodo antenato che è un figlio sinistro di un nodo y.
- Il nodo y è il nodo cercato, il successore

- Si mantengono pertanto i puntatori a due generici antenati x e y e si risale fino a quando x=left[y]

Visualizzazione



PseudoCodice Successore

```
Tree-Successore(x)
1  if right[x] ≠ NIL
2  thenreturn Tree-Minimum(right[x])
3  y ← p[x]
4  while y ≠ NIL e x = right[y]
5  do  x ← y
6      y ← p[x]
7  return y
```

Predecessore

- La procedura per la determinazione del predecessore è simmetrica a quella vista per il successore
- il predecessore si troverà nel sottoalbero sinistro (se questo esiste), e sarà l'elemento massimo di questo sottoalbero
- se non esiste sottoalbero sinistro il predecessore sarà l'antenato più prossimo che ha un figlio destro che è antenato del nodo in questione

PseudoCodice Predecessore

```
Tree-Predecessore(x)
1  if left[x] ≠ NIL
2  then return Tree-Maximum(left[x])
3  y ← p[x]
4  while y ≠ NIL e x = left[y]
5  do   x ← y
6       y ← p[x]
7  return y
```

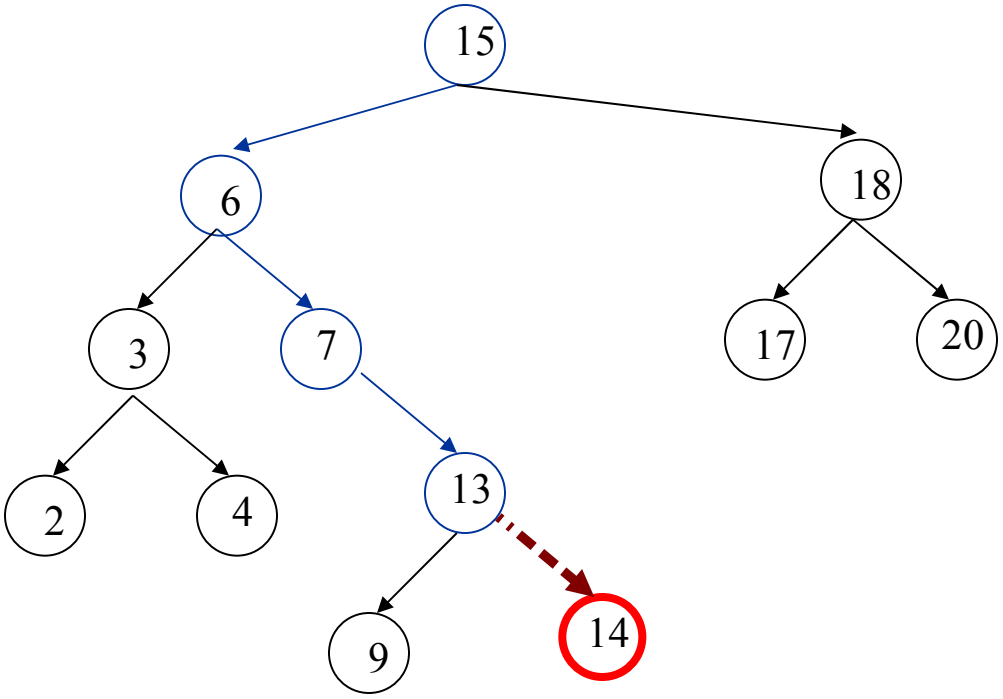
Inserzione

- Per inserire un nuovo valore k in un albero binario di ricerca, si prepara un nodo z tale che:
  - possieda come chiave key[z]=k
  - e non abbia collegamenti left[z]=right[z]=p[k]=NIL
- si cerca la posizione in cui inserirlo
- si modificano i campi di z per allacciarlo all'albero binario di ricerca

Inserzione

- Per trovare la posizione giusta ci si muove a partire dalla radice spostandosi sul sottoalbero destro o sinistro come in una ricerca
- si prosegue però fino ad arrivare ad un punto in cui fallirebbe la ricerca
- a questo punto si inserisce il nuovo nodo

Visualizzazione: inserzione di key 14



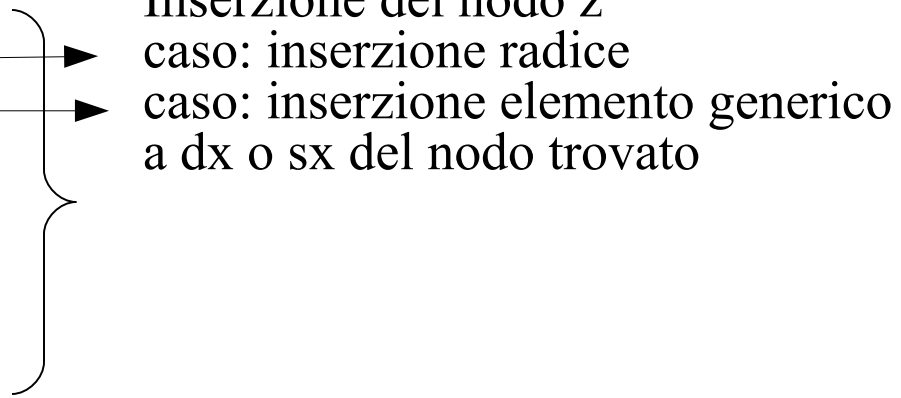
Pseudocodice Inserzione

```
Tree-Insert(T,z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4  do   y ← x
5       if key[z]<key[x]
6       then x ← left[x]
7       else x ← right[x]
8  p[z] ← y
9  if y=NIL
10 then root[T] ← z
11 else if key[z] < key[y]
12      then   left[y] ← z
13      else   right[y] ← z
```

ricerca della posizione

Pseudocodice Inserzione

```
Tree-Insert(T,z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4  do   y ← x
5       if key[z]<key[x]
6       then x ← left[x]
7       else x ← right[x]
8  p[z] ← y
9  if y=NIL
10 then root[T] ← z
11 else if key[z] < key[y]
12      then   left[y] ← z
13      else   right[y] ← z
```

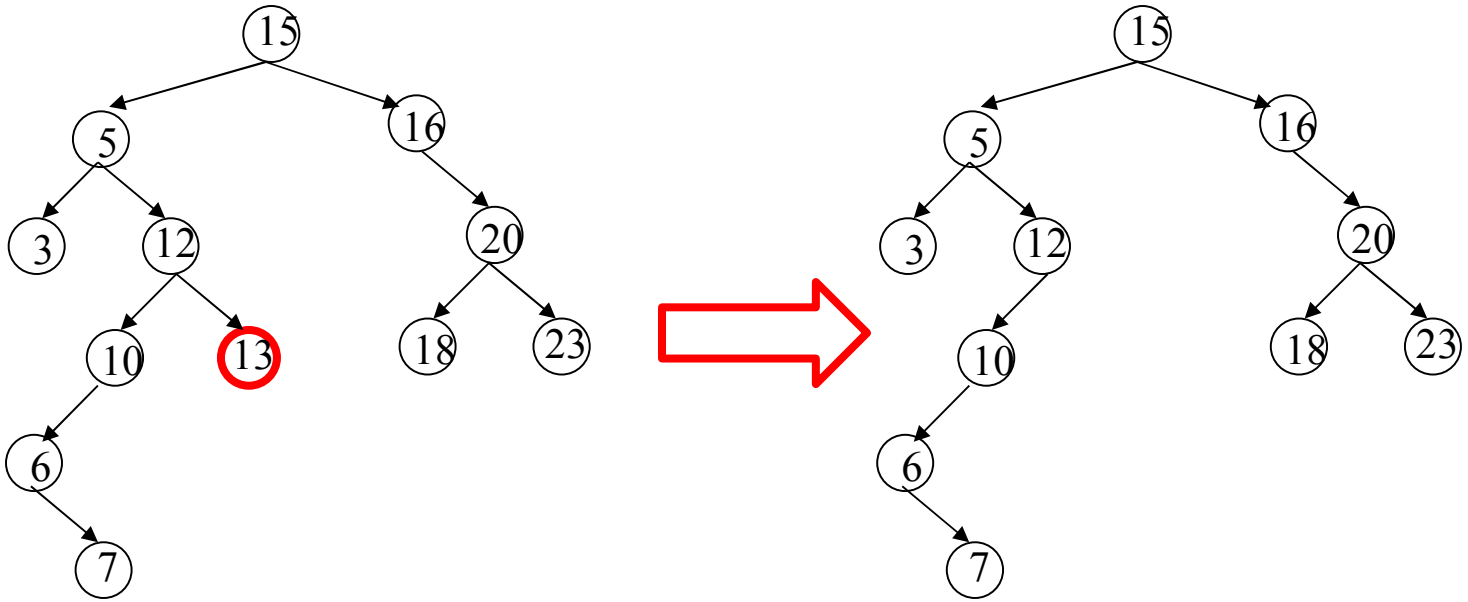


Cancellazione

- La procedura di cancellazione è più laboriosa in quanto si deve tenere conto di tre casi possibili
- dato un nodo z i casi sono:
  - z non ha figli
  - z ha un unico figlio
  - z ha due figli

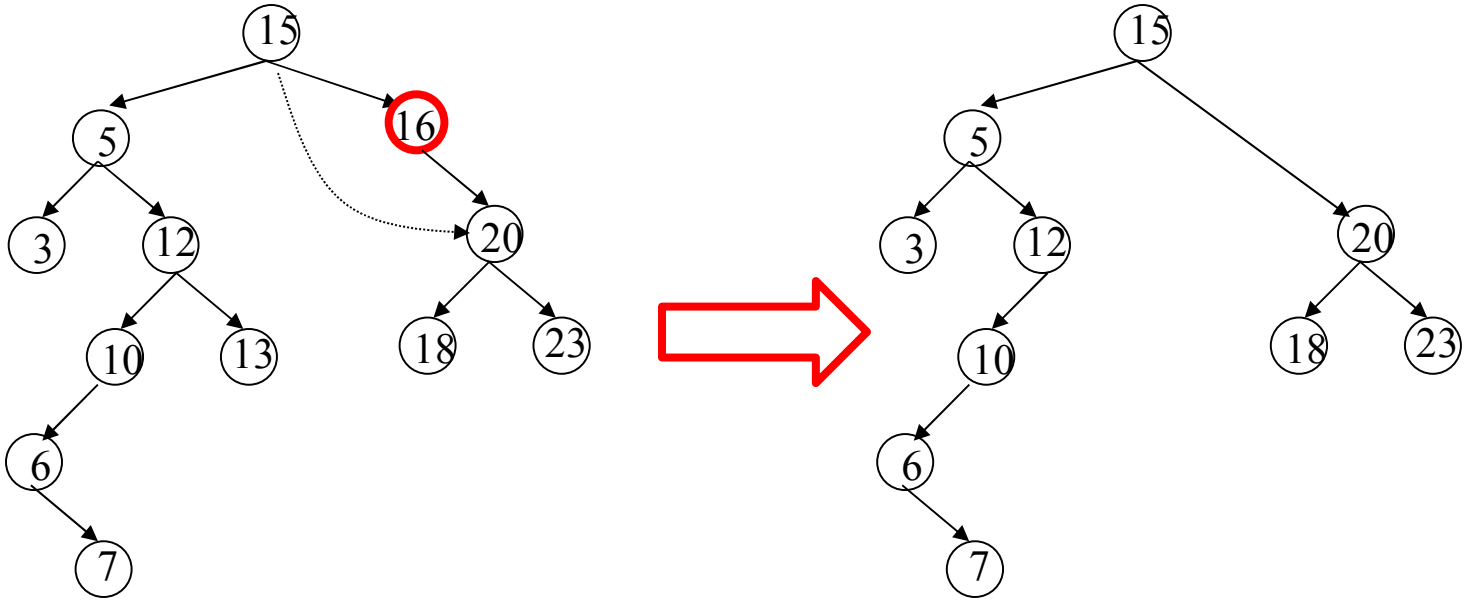
Caso 1:

- Nel caso in cui z non abbia figli si elimina direttamente il nodo z



Caso 2:

- Nel caso in cui z abbia un unico figlio si rimuove z e si collega il figlio al posto di z
- il secondo caso è identico al caso di eliminazione di un nodo da una lista concatenata

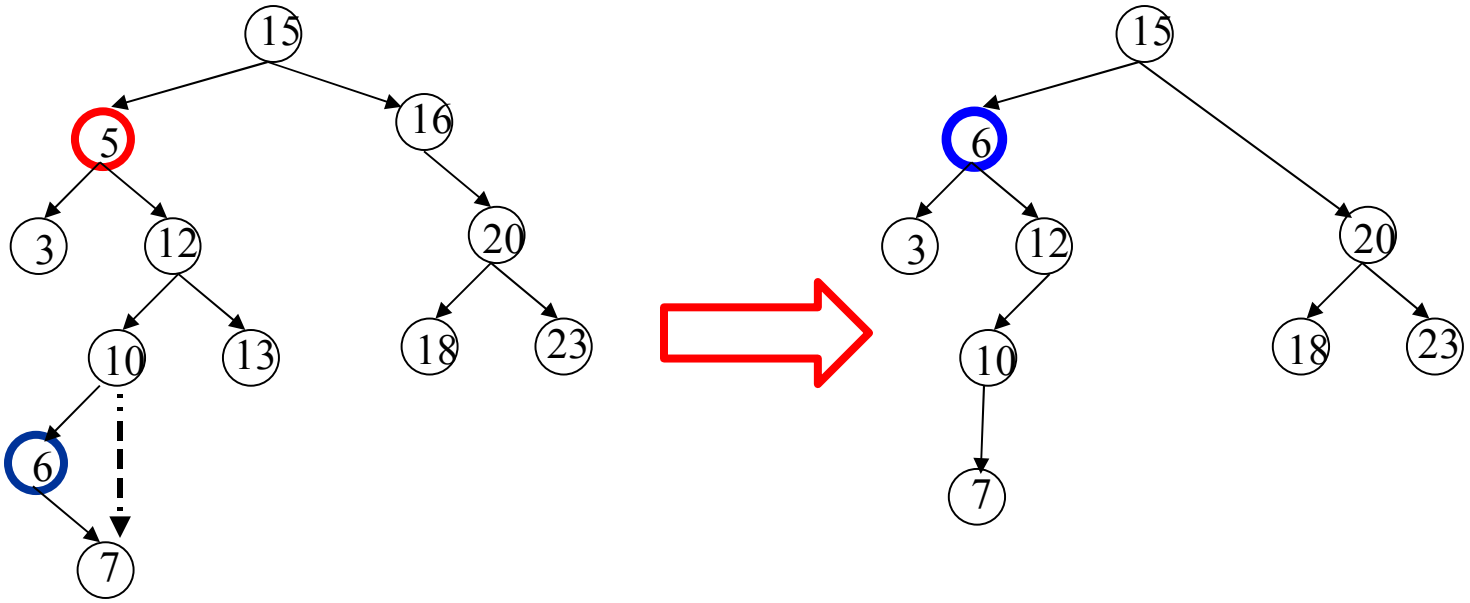


Caso 3:

- Se il nodo x da eliminare ha due figli si procede trovando un sostituto per x
- infatti non possiamo semplicemente eliminarlo e riconnettere i suoi figli perche' il padre di x potrebbe avere un altro figlio oltre a x ed avrebbe cosi' tre figli!!
- quale nodo e' un buon candidato per la sostituzione? deve essere un nodo che non obbliga a spendere molto sforzo per riaggiustare l'albero binario di ricerca
- il miglior candidato e' il successore:
  - il figlio sx di X e' anche minore del successore di X
  - il figlio dx di X e' anche maggiore del successore di X

Caso 3:

- Nel caso in cui z abbia 2 figli si determina il successore x di z
- si copia il contenuto di x al posto di quello di z
- infine si elimina il vecchio nodo x (caso 1 o 2)
  - Nota: il successore di x non può avere 2 figli perché non può avere un figlio sx (altrimenti sarebbe questo il successore)



PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Determinazione del nodo da cancellare caso 1 o 2

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Determinazione del nodo da cancellare
caso 3
y contiene il nodo da cancellare

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

in x si memorizza l'unico figlio
del nodo da cancellare (può essere NIL)

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Se x esiste si attacca all'albero

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Se il nodo cancellato era la radice allora la nuova radice è x

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Se il nodo cancellato era un figlio sx allora si aggiorna il puntatore sx altrimenti dx

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Se il nodo cancellato ha richiesto la determinazione del successore allora copia la chiave del successore

PseudoCodice Cancellazione

```
Tree-Delete(T,z)
1  if left[z]=NIL o right[z]=NIL
2  then y ← z
3  else y ← Tree-Successor(z)
4  if left[y] ≠ NIL
5  then x ← left[y]
6  else x ← right[y]
7  if x ≠ NIL
8  then p[x] ← p[y]
9  if p[y] = NIL
10 then root[T] ← x
11 else if y=left[p[y]]
12      then    left[p[y]] ← x
13      else    right[p[y]] ← x
14 if y ≠ z
15 then key[z] ← key[y]
17 return y
```

Restituisci il nodo cancellato per una eventuale deallocazione

Esercizio

- Implementare un albero binario di ricerca:
  - Classe Nodo
  - Classe Albero
  - Ricerca
  - Massimo/Minimo
  - Inserzione/cancellazione
  - Stampa

Implementazione C++

```
#include<iostream>
#include<cassert>
using namespace std;

template<class T, class  LessClass >
class TreeClass{
friend ostream& operator<<(ostream& out, TreeClass<T,LessClass>& aT);
private:
  class Node{
    public:
    Node(T aKey):left(0),right(0),parent(0),key(aKey){}
    Node *left, *right, * parent;
    T key;
    };
  Node *root;
public:
  TreeClass():root(0){}
  void insert(T);
  void print(ostream& out){p_print(root,out);}
  bool search(T user_key){return p_search(root, user_key);}
  T minimum();
  T maximum();
private:
  void p_print(Node *,ostream&);
  bool p_search(Node * x, T user_key);
};
```

Implementazione C++

```
template<class T, class  LessClass >
void TreeClass<T, LessClass>::p_print(Node *x, ostream& out){
  if(x!=0){
    p_print(x->left,out);
    out<<x->key<<" ";
    p_print(x->right,out);
  }
}

template<class T, class  LessClass >
bool TreeClass<T, LessClass>::p_search(Node * x, T usr_key){
  LessClass less;
  if(x==0) return false;
  if(!less(x->key,usr_key) && !less(usr_key,x->key))
   return true; //ugualianza
  if(less(usr_key,x->key)) p_search(x->left, usr_key);
  else p_search(x->right, usr_key);
}
```

```
template<class T, class LessClass >
void TreeClass<T,LessClass>::insert(T usr_key){
  LessClass less;
  //inizializzazione del nodo da aggingere
  Node* z=new Node;
  assert(z);
  z->key=usr_key; z->left=0; z->right=0; z->parent=0;
  //ricerca della giusta posizione di inserzione
  Node* y=0;
  Node* x=root;
  while(x != 0){
    y=x;
    if(less(z->key, x->key)) x=x->left;
    else x=x->right;
  }
  //settaggio dei puntatori
  z->parent=y;
  if(y==0) root=z;
  else if(less(z->key, y->key)) y->left=z;
  else y->right=z;
}
```

Implementazione C++

```
template<class T, class  LessClass >
T TreeClass<T, LessClass>::minimum(){
  assert(root);
  Node* x=root;
  while(x->left !=0) x=x->left;
  return x->key;
}

template<class T, class  LessClass >
T TreeClass<T, LessClass>::maximum(){
  assert(root);
  Node* x=root;
  while(x->right !=0) x=x->right;
  return x->key;
}

template<class T, class  LessClass >
ostream& operator<<(ostream& out, TreeClass<T,LessClass>& aT)
{
    aT.print(out);
    return out;
}
```

Implementazione C++

```
template<class T> struct LessClass{
  bool operator()(const T & a, const T & b)const{return a<b;}
};

int main(){
  //integer example
  int v[]={2,5,8,1,3,4,7,9,6,0};
  TreeClass<int, LessClass<int> > T;

  for(int i=0;i<10;i++) T.insert(v[i]);
  cout<<T<<endl;

  cout<<"Searching 5:"<<
  (T.search(5)? "Found":"Not found")<<endl;
  cout<<"Searching 11:"<<
  (T.search(11)? "Found":"Not found")<<endl;
  cout<<"Searching maximum:"<<T.maximum()<<endl;
  cout<<"Searching minimum:"<<T.minimum()<<endl;
```

Implementazione C++

```
  //char example
  char c[]="this_is_a.";
  TreeClass<char, LessClass<char> > Tc;

  for(int i=0;i<10;i++)
    Tc.insert(c[i]);
  cout<<Tc<<endl;

  cout<<"Searching s:"<<
  (Tc.search('s')? "Found":"Not found")<<endl;
  cout<<"Searching v:"<<
  (Tc.search('v')? "Found":"Not found")<<endl;
  cout<<"Searching maximum:"<<Tc.maximum()<<endl;
  cout<<"Searching minimum:"<<Tc.minimum()<<endl;

  cout<<endl;
  return 0;
}
```