Binary Search Tree

Corso: Strutture Dati

Docente: Annalisa De Bonis

Alberi di ricerca binari (BST)

- Un albero di ricerca binario è un albero binario che memorizza in ciascun nodo una chiave in modo tale che
  - Se u, v e w sono tre nodi interni tali che u si trova nel sottoalbero sinistro di v e w si trova nel sottoalbero destro di v, allora

    key(u) ≤ key(v) ≤ key(w)

  (sulle chiavi è definita una relazione di ordine totale)

Operazioni supportate da un BST

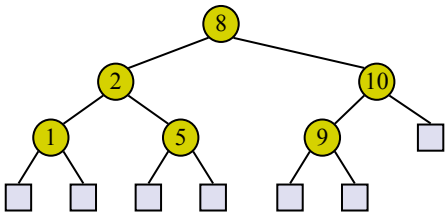
- Cancellazione
- Inserimento
- Ricerca di una chiave
- Ricerca del minimo
- Ricerca del massimo
- Ricerca del successore
- Ricerca del predecessore

Alberi di ricerca binari (BST)

assunzioni usate nelle implementazione

- nelle nostre implementazioni, assumeremo che
  - Un albero di ricerca binario è un albero binario proprio
    - *Un albero binario proprio è un albero in cui ogni nodo o è una foglia o ha due figli*
  - Le chiavi sono memorizzate solo nei nodi interni
    - Nelle implementazioni del TDA Dictionary assumeremo che i nodi interni contengono coppie del tipo (chiave,valore)
  - Le foglie non contengono niente

Esempio



Visita inorder di un BST

- Una visita inorder di un albero di ricerca binario visita le chiavi in ordine crescente
  - Possiamo ottenere la sequenza ordinata della chiavi

Algoritmo di ricerca

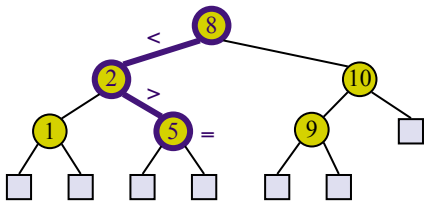
- Input: la chiave da cercare k e un nodo v del BST
- Output: un nodo w del sottoalbero radicato in v, tale che o w è un nodo interno contenente k o w è una foglia che si trova nel posto in cui si troverebbe k se appartenesse al sottoalbero

O(altezza)

```
Algorithm TreeSearch(k, v)
    if T.isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, T.left(v))
    else if k = key(v)
        return v
    else          // k > key(v)
        return TreeSearch(k, T.right(v))
```

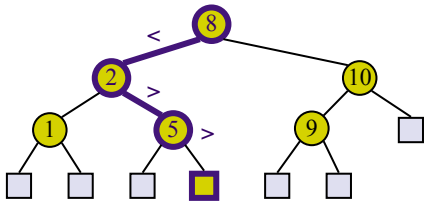

Esempio

- TreeSearch(5,root)



Esempio

- TreeSearch(6,root)



Algoritmo di Inserimento

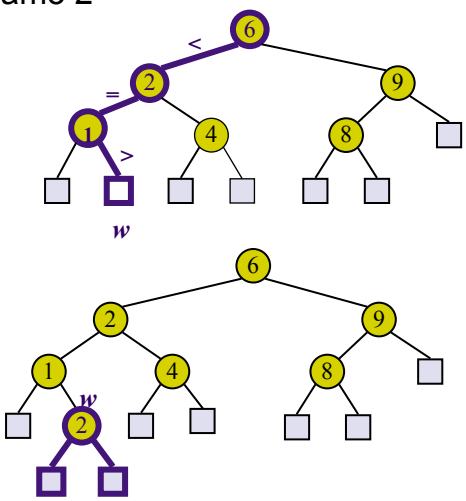
- Input: una chiave k, un valore x, un nodo v
- Output: un nuovo nodo del sottoalbero radicato in v che contiene la coppia (k,x)

```
Algorithm TreeInsert(k, x,v)
  w ← TreeSearch(k, v)                                   O(altezza)
   if T.isInternal(w)
     return TreeInsert(k, x,T.left(w))  // inserisce nel sottoalbero sinistro di v
                                        //(sarebbe andato bene anche il destro)
   T.insertAtExternal(w,(k,x)) // w in questo caso è la foglia in cui inserire (k,x)
  return w
```

- InsertAtExternal(w,e) inserisce e nella foglia w e trasforma w in un nodo interno avente come figli due foglie nuove. Se v è un nodo interno allora si verifica un errore.

Esempio

- Inseriamo 2



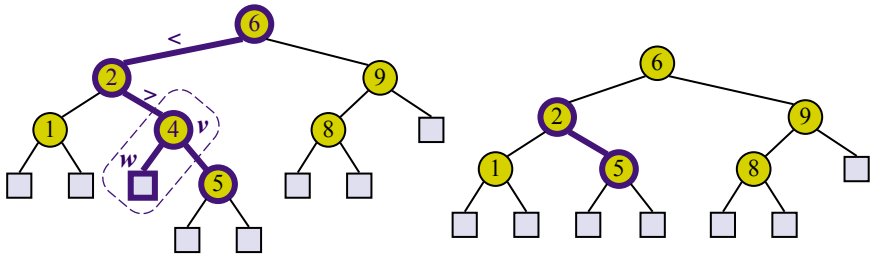
Algoritmo di cancellazione

- Input: la chiave k da rimuovere
- Output: se k è nell'albero, restituisce l'entrata con chiave k dopo averla cancellata; altrimenti restituisce **null**

- L'algoritmo remove(k) funziona come segue:
  - Se k è contenuta in un nodo interno v allora distinguiamo i due casi seguenti:
    - Uno dei figli di v è una foglia
    - Entrambi I figli di v sono nodi interni

Algoritmo di cancellazione

- Caso 1: il nodo v da cancellare ha un figlio w che è una foglia
  - L'algoritmo di cancellazione rimuove i nodi v e w invocando removeExternal(*w*)
  - removeExternal(*w*) rimuove la foglia w e il padre di w dall'albero e sostituisce il padre di w con il fratello di w; se w non è una foglia si ha un errore.
  - Esempio: remove(4)

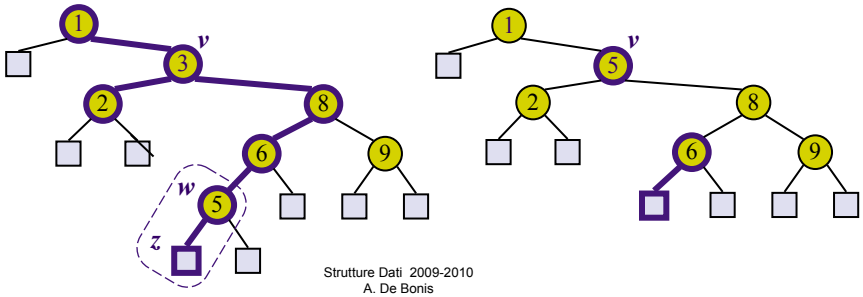


Algoritmo di cancellazione

- Caso 2: la chiave da cancellare è contenuta in un nodo v i cui figli sono entrambi nodi interni
  - Troviamo il primo nodo interno w che segue v nella visita inorder
  - Copiamo key(w) nel nodo v
  - Rimuoviamo w e il suo figlio sinistro z (che è una foglia) mediante removeExternal(z)

Il tempo di esecuzione è O(altezza)

- Esempio: remove(3)



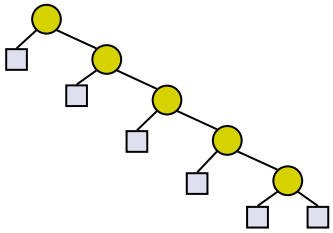
Dizionari implementati come BST

- Possiamo implementa il TDA Dictionary mediante un albero di ricerca binario
  - Ogni entrata viene memorizzata in un nodo interno
  - C'è bisogno di un comparatore per confrontare le chiavi

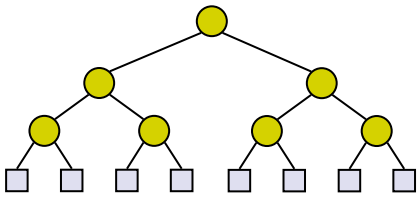
Complessità dei medodi di Dictionary

- Consideriamo un dizionario con n entrate implementato con un BST di altezza h
  - Lo spazio usato è $O(n)$
  - I metodi find, insert e remove impiegano tempo $O(h)$
- Nel caso pessimo $h = O(n)$; nel caso ottimo h= $O(\log n)$

Caso pessimo



Caso ottimo



La classe BinarySearchTree

```
public class BinarySearchTree<K,V>
 extends LinkedBinaryTree<Entry<K,V>> implements Dictionary<K,V> {

 protected Comparator<K> C;
 protected int numEntries = 0;

public BinarySearchTree()  {
   C = new DefaultComparator<K>();
   addRoot(null);
 }

 public BinarySearchTree(Comparator<K> c)  {
   C = c;
   addRoot(null);
 }
```


La classe BinarySearchTree: la classe innestata BSTEntry

```
protected static class BSTEntry<K,V> implements
    Entry<K,V> {
    protected K key;
    protected V value;
    protected Position<Entry<K,V>> pos;
    BSTEntry() {}
    BSTEntry(K k, V v, Position<Entry<K,V>> p) {
      key = k; value = v; pos = p;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public Position<Entry<K,V>> position() { return pos; }
 }
```

La classe BinarySearchTree: il metodo checkEntry

```
//metodo ausiliario per controllare un' entrata
protected void checkEntry(Entry<K,V> ent) throws
    InvalidEntryException {
  if(ent == null || !(ent instanceof BSTEntry))
    throw new InvalidEntryException("entrata non
    valida");
}
```


La classe BinarySearchTree: il metodo checkKey

```
//metodo ausiliario per controllare una chiave
protected void checkKey(K key) throws
    InvalidKeyException {
  if(key == null) throw new InvalidKeyException("chiave
    uguale a null");
   try { C.compare(key,key); }
    catch(Exception e) { throw new InvalidKeyException
    ("chiave non valida"); }

}
```

La classe BinarySearchTree: il metodo treeSearch

```
//metodo ausiliario usato dai i metodi find, remove e insert di Dictionary
protected Position<Entry<K,V>> treeSearch(K key, Position<Entry<K,V>>
    pos) {
 if (isExternal(pos)) return pos;
 else {
   K curKey = key(pos);
   int comp = C.compare(key, curKey);
   if (comp < 0)
     return treeSearch(key, left(pos));
   else if (comp > 0)
     return treeSearch(key, right(pos));
   return pos;
  }
}
```


La classe BinarySearchTree: il metodo removeExternal

```
//metodo ausiliario per rimuovere la foglia v e il padre di v
protected void removeExternal(Position<Entry<K,V>> v)
                  throws InvalidPositionException {
  if (!isExternal(v)) throws new InvalidPositionException("il nodo non è
    una  foglia")
  BTNode p = (BTNode) parent(v);
  BTNode s = (BTNode) sibling(v);
  if (isRoot(p))                          //se il padre di v è la radice
    { s.setParent(null); root = s; }  //allora il fratello di v diventa radice
  else                    //altrimenti il fratello di v diventa figlio del "nonno"
       { BTNode g = (BTNode) parent(p);
         if (p == leftChild(g)) g.setLeft(s);
         else g.setRight(s);
         s.setParent(g); }
    numEntries -= 1;
}
```

La classe BinarySearchTree: il metodo remove

```
public Entry<K,V> remove(Entry<K,V> ent) throws
  InvalidEntryException  {
  checkEntry(ent);
  Position<Entry<K,V>> remPos =
                              ((BSTEntry<K,V>) ent).position();
  Entry<K,V> toReturn = ent;
  //caso 1: uno dei due figli è una foglia
  if (isExternal(left(remPos))) remPos = left(remPos);
  else if (isExternal(right(remPos))) remPos = right(remPos);
```

Continua nella prossima slide

La classe BinarySearchTree: il metodo remove

```
//caso 2: entrambi i figli sono nodi interni
else {
   Position<Entry<K,V>> swapPos = remPos;
   // cerca il successore di ent
   remPos = right(swapPos);
    do
      remPos = left(remPos);
    while (isInternal(remPos)); //si ferma quando remPos e` una foglia
    replaceEntry(swapPos, (Entry<K,V>) parent(remPos).element());
  }
  removeExternal(remPos);
  return toReturn;
}
```

La classe BinarySearchTree: il metodo insert

```
public Entry<K,V> insert(K k, V x) throws
    InvalidKeyException {
    checkKey(k);
   Position<Entry<K,V>> insPos = treeSearch(k, root());
    while (!isExternal(insPos))
       insPos = treeSearch(k, left(insPos));
   return insertAtExternal(insPos, new BSTEntry<K,V>(k, x,
    insPos));
 }
```


La classe BinarySearchTree: il metodo insertAtExternal

```
//metodo ausiliario per inserire un'entrata in una foglia
protected Entry<K,V> insertAtExternal
    (Position<Entry<K,V>> v, Entry<K,V> e) {
  expandExternal(v,null,null);
  replace(v, e);
  numEntries++;
  return e;
 }
```

La classe BinarySearchTree: il metodo expandExternal

```
//il metodo ausiliario di LinkedBinaryTree che espande una //
foglia in un  nodo interno avente come figli due foglie con //
elementi l ed r risp.
public void expandExternal(Position<E> v, E l, E r)
   throws InvalidPositionException {
   if (!isExternal(v))
     throw new InvalidPositionException("il nodo non e` una
                 foglia");
   insertLeft(v, l);
   insertRight(v, r);

}
```


Altri metodi ausiliari di BinarySearchTree

- Oltre ai metodi di Dictionary, la classe BinarySearchTree deve contenere **anche** i seguenti metodi ausiliari:
  - protected K key(Position<Entry<K,V>> position)
  - Protected V value(Position<Entry<K,V>> position)
  - protected Entry entry(Position<Entry<K,V>> position)
  - protected void replaceEntry(Position <Entry<K,V>> pos, Entry<K,V> ent)
  - protected void addAll(PositionList<Entry<K,V>> L, Position<Entry<K,V>> v, K k)

Esercizi

- Completare la classe BinarySearchTree
- Scrivere un programma di test per la classe BinarySearchTree che testi tutti i metodi dell'interfaccia Dictionary