Programmazione 2
e Laboratorio di Programmazione

Corso di Laurea in

Informatica

Università degli Studi di Napoli "Parthenope"

Anno Accademico 2023-2024

Prof. Luigi Catuogno

1

Informazioni sul corso

Docente	Luigi Catuogno **luigi.catuogno@uniparthenope.it**
Orario	Lun: 9:00-11:00 Mer: 11:00-13:00
Sede	Centro Direzionale Napoli **Aula Magna**
Ricevimento	Mer: 14:00-16:00 (previo appuntamento) Ufficio docente oppure Team: **cxxa3bo**

2

Libri di testo

Introduzione al linguaggio – costrutti e tecniche di base

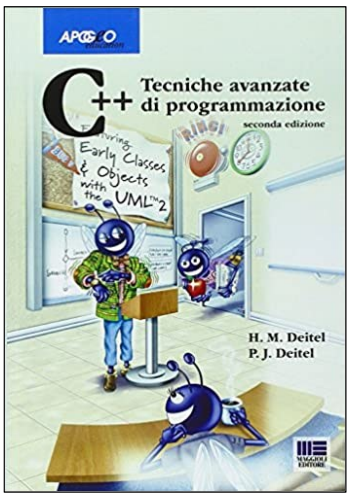
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

3

Libri di testo

Tecniche avanzate e strutture dati elementari

[TAP] H. M. Deitel, P. J. Deitel
C++ Tecniche avanzate di programmazione
II ed. (2011) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8572-6



4

Risorse on-line

Team del corso

Programmazione 2 AA 2023-24 - Prof. Catuogno
Comunicazioni, incontri e avvisi per il corso
Codice: **ftomzjx**

Piattaforma e-learning

Programmazione II e Laboratorio di Programmazione II - A.A. 2023-24
Materiale didattico, manualistica, esercitazioni.
URL: https://elearning.uniparthenope.it/course/view.php?id=2386

5

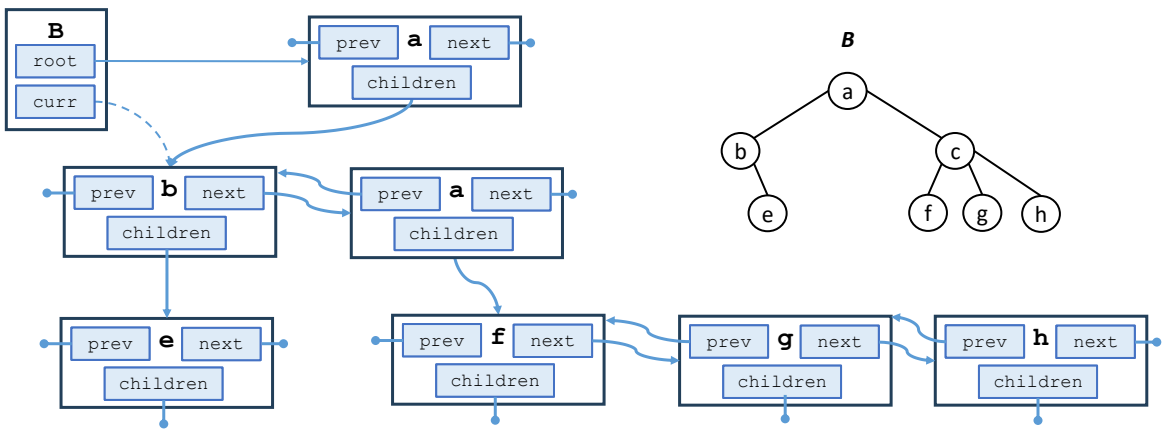
Strutture dati elementari

6

Implementazioni degli alberi

7

Esempio: albero con *«liste di figli»*



8

Implementazione degli alberi con gli array

- Per alcune applicazioni, può essere conveniente implementare gli alberi utilizzando gli array. Le diverse tecniche rientrano nel novero delle *«rappresentazioni indicizzate»*
- L'idea di base è rappresentare ogni nodo dell'albero ***T (N,A)*** con una cella dell'array **`P[]`** ***;***
  - I nodi di ***N*** *«mappano»* l'indice della cella dell'array che li rappresenta - *e.g.* se i nodi sono rappresentati da *interi positivi,* il nodo *v* di ***N*** sarà rappresentato dalla cella **`P[v]`**;
  - Ciascuna cella contiene un record con tutte le informazioni immagazzinate nel nodo (dati, chiavi…) e *possibilmente* le informazioni necessarie per raggiungere i nodi genitori o figli.

9

Implementazione degli alberi con gli array

- Le tecniche impiegate per esplorare l'albero variano a seconda del tipo di rappresentazione dell'albero e delle sue proprietà. Possono essere definiti diversi attributi e metodi:
  - **`P[v].parent`** che restituisce l'indice del nodo padre di v (root ha padre nullo);
  - **`figlio(v)`** restituisce la posizione del primo figlio di v (se c'è) a partire dalla posizione corrente dell'array.
  - **`P[v].data`** restituisce i dati contenuti nel nodo *v*
  - **`P[v].key`** rappresenta la *chiave di ricerca* del nodo *v*

10

Implementazione degli alberi con gli array

- Due delle più comuni rappresentazioni di alberi basate su array:
  - *Vettore dei padri*
  - *Vettore posizionale*

11

Vettore dei padri

Sia ***T*=*(N,A)*** un albero con *n* nodi numerati da *0* a *(n-1)*

- Un *vettore dei padri* è un array **`P[]`** di dimensione *n* in cui le celle contengono la coppia *(item, parent)*
- **`P[v].parent=u`** se e solo se, c'è un arco *(u,v)* in ***A***.
- Se *v* è la radice **`P[v].parent=null`**

12

Vettore dei padri

- La rappresentazione con *vettore dei padri* permette di identificare il genitore di un nodo in tempo *costante:* $O(1)$
- Per identificare i figli di un nodo *v*, è necessario scorrere l'array e cercare tutti i nodi *i* in cui **`P[i].parent=v`**; Il tempo impiegato è proporzionale al numero di nodi nell'albero: $O(n)$, dove n è il numero di nodi nell'albero.

13

Vettore posizionale

- Rappresentazione utile nel caso particolari di alberi *d*-ari completi (o quasi completi)
- Ogni nodo ha una posizione prestabilita nell'array.

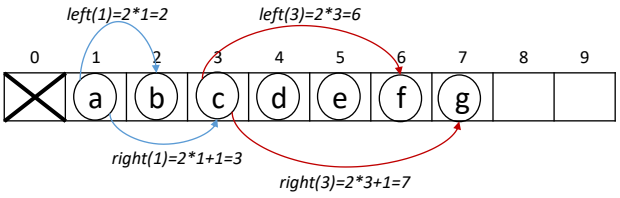
14

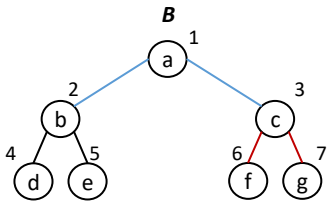
Vettore posizionale

- Sia ***T*=*(N,A)*** un albero *d*-ario, con *n* nodi, *completo* o *quasi completo* i cui nodi sono numerati da *1* a *n.* Sia *d* il grado di ***T***
- Un *vettore posizionale* è un array **`P[]`** di dimensione *n* in cui le celle contengono i dati associati ai nodi (*item, key…)*
  - Dato un nodo *v*, i suoi dati sono nella posizione **`P[v]`**
  - L'*i*-esimo figlio di *v* si trova in posizione **`P[d*v+i]`**, con *i* compreso tra *0* e *d-1*;

15

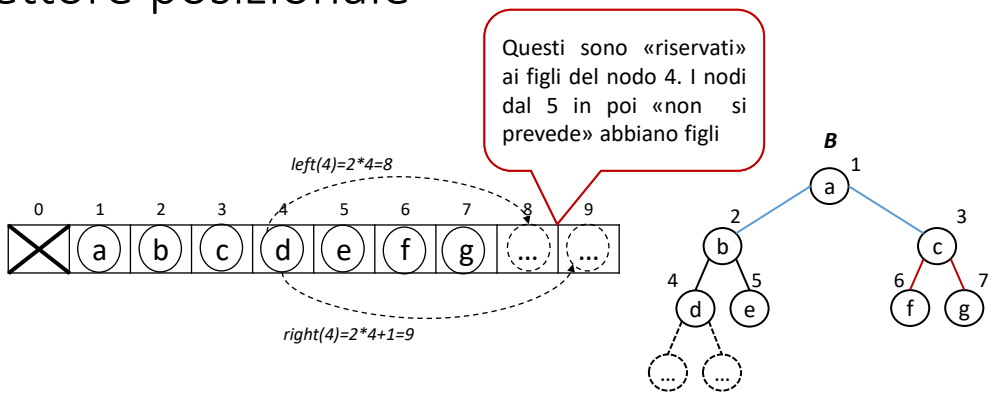
Vettore posizionale





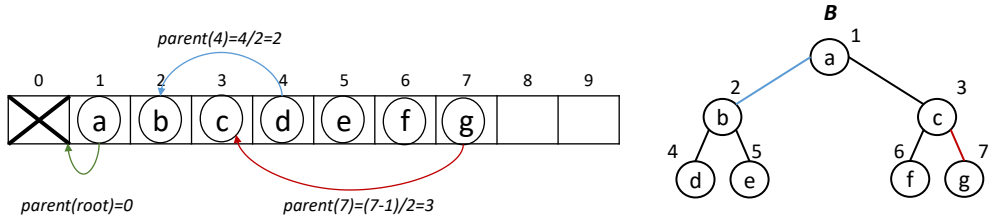
16

Vettore posizionale



17

Vettore posizionale



18

Esempio: albero binario con array

File: **binTreeArray.hpp**

```
14  class binTreeArray {
15          int *storage;
16          bool *isNull;
17          int size;
18  public:
19          binTreeArray(int n) {
20                  storage = new int[n+1];
21                  isNull = new bool[n+1];
22                  for(int i=0;i<=n;i++)
23                          isNull[i]=true;
24                  size=n;
25          }
26          ~binTreeArray() {
27                  delete [] storage;
28                  delete [] isNull;
29          }
 …  …
```

19

Esempio: albero binario con array

File: **binTreeArray.hpp**

```
31   void insert(int n,int data) {
32                  if (n<1||n>size||!isNull[n])
33                          throw insertException();
34                  storage[n]=data;
35                  isNull[n]=false;
36          }
37
38          int data(int n) {
39                  if (n<1||n>size||isNull[n])
40                          throw nodeNotFound();
41                  return storage[n];
42          }
 …  …
```

20

Esempio: albero binario con array

File: **binTreeArray.hpp**

```
43          int left(int n) {
44                  if (n<1||n>size||isNull[n])
45                          throw nodeNotFound();
46                  return 2*n;
47          }
48          int right(int n) {
49                  if (n<1||n>size||isNull[n])
50                          throw nodeNotFound();
51                  return 2*n+1;
52          }
53          int parent(int n) {
54                  if (n<1||n>size||isNull[n])
55                          throw nodeNotFound();
56                  if(n%2)
57                          return (n-1)/2;
58                  else
59                          return n/2;
60          }
```

21

Esempio: albero binario con array

File: **binTreeArray.hpp**

```
61          int hasLeft(int n) {
62                  …
63                  return !isNull[2*n];
64          }
65
66          int hasRight(int n) {
67                  …
68                  return !isNull[2*n+1];
69          }
70          int isLeaf(int n) {
71                 return !(hasLeft[n]||hasRight[n]);
72          }
 …  …
80          void inorderTrav();
90          void preorderTrav();
91          void postorderTrav();
```

22

Esempio: albero binario con array

File: **binTreeArray.hpp**

```
100         void removeSubtree();
101         void doRemoveSubtree(int);
102
103 private:
104         do_inorderTrav(int);
105         do_preorderTrav(int);
106         do_postorderTravi(int);
107 };
108
109
110
111
112
113
114
115
```


Esercizio: albero binario con array

File: **binTreeArray.cpp**

Implementare i metodi

```
void inorderTrav();
void preorderTrav();
```

per la classe **binTreeArray.**

Esercizio: albero binario con array

File: **binTreeArray.hpp**

```
76  void binTreeArray::do_inorderTrav(int i) {
77          if(hasLeft(i))
78                  do_inorderTrav(left(i));
79          cout<<data(i);
80          if(hasRight(i))
81                  do_inorderTrav(right(i));
82  }
83
84  void binTreeArray::inorderTrav() {
85          do_inorderTrav(0);
86  }
 …  …
```

25

Esercizio: albero binario con array

File: **binTreeArray.cpp**

Implementare il metod

`void removeSubtree(int n);`

per la classe **`binTreeArray.`**

26

Esercizio: albero binario con array

File: **binTreeArray.hpp**

```
 90  void binTreeArray::do_removeSubtree(int n) {
 91          if(hasLeft(n))
 92                  do_removeSubtree(left(n));
 93          if(hasRight(n))
 94                  do_removeSubtree(right(n));
 95          isNull[n]=true;
 96  }
 97
 98  void binTreeArray::removeSubtree() {
 99          if(!isNull[1])
100                  do_deleteSubtree(1);
101  }
 …  …
```

27

Vettore posizionale

- La rappresentazione con *vettore posizionale* permette di identificare il genitore di un nodo in tempo *costante:* $O(1)$
- Il figlio *i-esimo* di un nodo *v può* essere indicizzato in tempo costante: $O(1)$
- L'enumerazione dei figli di un nodo *v*, può essere effettuata in tempo $O(d)$ impiegato è proporzionale al grado nell'albero

28

Implementazione alberi: *array* vs *linked list*

- La rappresentazione mediante array:
  - È semplice ed efficiente
  - Permette di trarre vantaggio da certe proprietà degli alberi
  - Ridotto consumo di memoria e tempi «di gestione»
- Svantaggi
  - Poco flessibile (dimensioni statiche)
- Utile quando si vogliono utilizzare gli algoritmi sugli alberi, per manipolare sequenze di dati contenute in un array

29

Alberi binari di ricerca

30

Alberi binari di ricerca *(Binary Search Tree)*

- Un BST è una struttura particolarmente efficiente per implementare insiemi *(set)* ordinati di elementi *distinti* .
- Tra i principali operatori:
  - aggiunta/eliminazione di elementi
  - visite
  - ricerca di elementi
  - estrazione degli elemento massimo e minimo
  - predecessore/successore di un elemento…

31

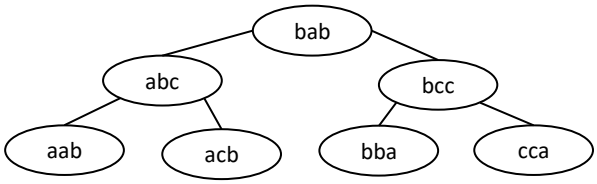
Alberi binari di ricerca *(Binary Search Tree)*

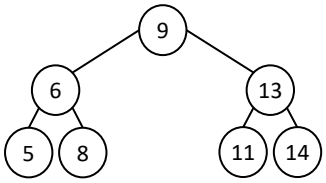
- Sia ***T*** = ***(N,A)*** un albero binario in cui ciascun nodo di ***N*** possieda un attributo detto *chiave* (*key*) e si assuma che nell'insieme dei valori delle chiavi ***K*** sia definita una relazione d'ordine totale (*e.g.* **<**)
- ***T*** è un *albero binario di ricerca* se:
  - Qualsiasi nodo appartenente al *sottoalbero sinistro* della radice ***r***, ha chiave di valore minore della chiave di ***r***;
  - La chiave della radice ***r*** ha valore minore della chiave di qualsiasi nodo appartenente al sottoalbero destro di ***r***;
  - I sottoalberi destro e sinistro di ***r***, sono a loro volta alberi binari di ricerca (va da sé che un nodo foglia è sempre un BST)

32

Alberi binari di ricerca

Esempi di BST:

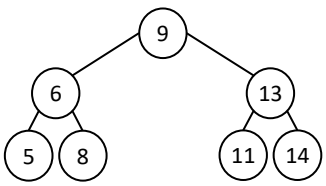




33

Alberi binari di ricerca

Se volessimo aggiungere un elemento con chiave=7?

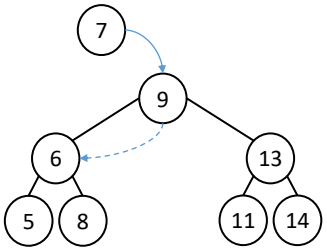


34

Alberi binari di ricerca

Se volessimo aggiungere un elemento con chiave=7?

Si parte dalla radice, si confrontano le chiavi: 7<9, quindi si va a sinistra…



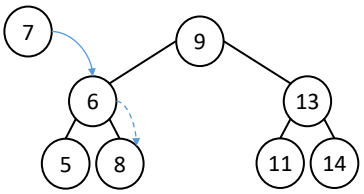
35

Alberi binari di ricerca

Se volessimo aggiungere un elemento con chiave=7?

Si parte dalla radice, si confrontano le chiavi: 7<9, quindi si va a sinistra…

Nuovo confronto: 7>6, si va a destra…



36

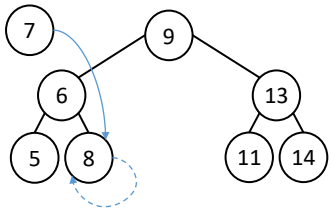
Alberi binari di ricerca

Se volessimo aggiungere un elemento con chiave=7?

Si parte dalla radice, si confrontano le chiavi: 7<9, quindi si va a sinistra…

Nuovo confronto: 7>6, si va a destra…

Nuovo contronto: 7<8 si andrebbe a sinistra ma…



37

Alberi binari di ricerca

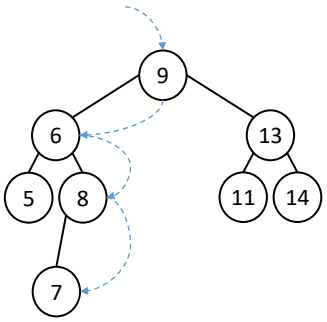
Se volessimo aggiungere un elemento con chiave=7?

Si parte dalla radice, si confrontano le chiavi: 7<9, quindi si va a sinistra…

Nuovo confronto: 7>6, si va a destra…

Nuovo contronto: 7<8 si andrebbe a sinistra ma…

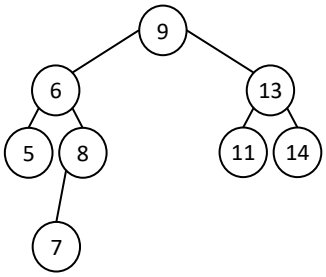
8 è una foglia, quindi 7 diventa il suo nodo sinistro.



38

Alberi binari di ricerca

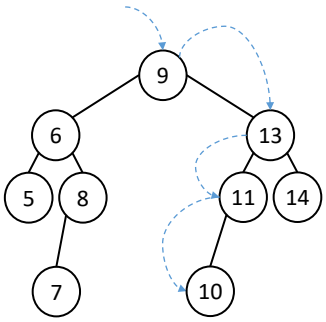
Inseriamo i nodi con chiave 10 e 2…



39

Alberi binari di ricerca

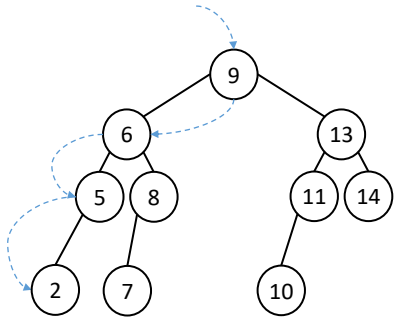
Inseriamo i nodi con chiave 10 e 2…



40

Alberi binari di ricerca

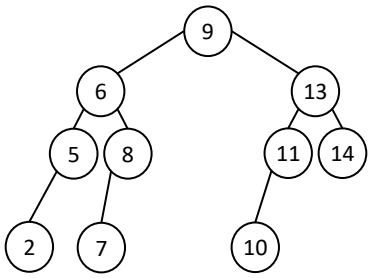
Inseriamo i nodi con chiave 10 e 2…



41

Alberi binari di ricerca

Una proprietà utile dei BST è che, la visita *inorder* dell'albero restituisce la sequenza ordinata delle chiavi (secondo la relazione d'ordine definita).



Visita: **2 4 6 7 8 9 10 11 13 14**

42

Esempio: Inserimento nodi in un BST

File: **binTreePtrBST.hpp**

```
…
23  binTreePtr *binTreePtr::BSTinsert(int key) {
24          if(isEmpty()) {
25                  insertRoot(key);
26                  return this;
27          }
28          doBSTinsert(root,key);
29          return this;
30  }
…
```

43

Esempio: Inserimento nodi in un BST

File: **binTreePtrBST.hpp**

```
5   void doBSTinsert(binTreePtrNode *n, int key) {
6           int item;
7           item=n->data;
8           if(item==key)
9                   throw keyFound();
10
11          if(key<item)
12                  if(n->hasLeft())
13                          doBSTinsert(n->left,key);
14                  else
15                          n->left=new binTreePtrNode(key);
16          else
17                  if(n->hasRight())
18                          doBSTinsert(n->right,key);
19                  else
20                          n->right=new binTreePtrNode(key);
21  }
```

Nella definizione abbiamo detto che i nodi sono tutti distinti, quindi se il nodo già c'è, lanciamo una eccezione.

44

Esempio: Inserimento nodi in un BST

File: **binTreePtrBST.hpp**

```
 5  void doBSTinsert(binTreePtrNode *n, int key) {
 6          int item;
 7          item=n->data;
 8          if(item==key)
 9                  throw keyFound();
10
11          if(key<item)
12                  if(n->hasLeft())
13                          doBSTinsert(n->left,key);
14                  else
15                          n->left=new binTreePtrNode(key);
16          else
17                  if(n->hasRight())
18                          doBSTinsert(n->right,key);
19                  else
20                          n->right=new binTreePtrNode(key);
21  }
```

Se la il nodo da inserire ha chiave minore di quella della radice, si prosegue sul sottoalbero sinistro, se c'è…

Esempio: Inserimento nodi in un BST

File: **binTreePtrBST.hpp**

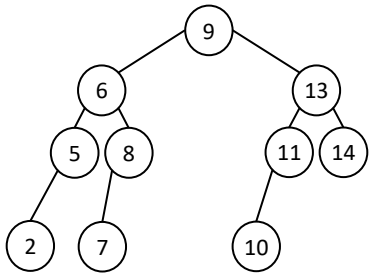
```
 5  void doBSTinsert(binTreePtrNode *n, int key) {
 6          int item;
 7          item=n->data;
 8          if(item==key)
 9                  throw keyFound();
10
11          if(key<item)
12                  if(n->hasLeft())
13                          doBSTinsert(n->left,key);
14                  else
15                          n->left=new binTreePtrNode(key);
16          else
17                  if(n->hasRight())
18                          doBSTinsert(n->right,key);
19                  else
20                          n->right=new binTreePtrNode(key);
21  }
```

… se non c'è, il nuovo nodo va aggiunto come figlio sinistro della radice.

… se il nuovo nodo ha chiave maggiore di quella della radice, si procede a destra in maniera del tutto analoga.

Ricerca in un BST

Per effettuare la ricerca di un nodo che abbia una chiave data: *e.g. key=10…*



47

Ricerca in un BST

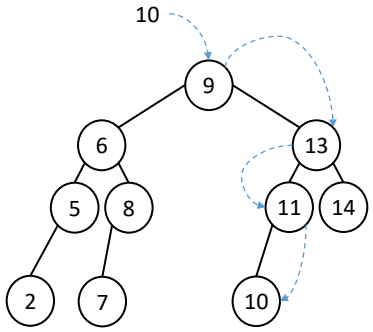
Per effettuare la ricerca di un nodo che abbia una chiave data: *e.g. key=10…*

Si parte dalla radice, si confrontano le chiavi: sono diverse, quella cercata è maggiore e quindi si va a destra…

Nuovo confronto: 10<13, si va a sinistra…

… 10<11, ancora a sinistra.

La ricerca termina con successo.



48

Ricerca in un BST

Per effettuare la ricerca di un nodo che abbia una chiave data: *e.g. key=4…*

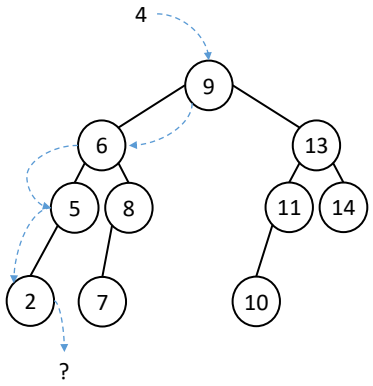
Si parte dalla radice, si confrontano le chiavi: sono diverse, quella cercata è minore e quindi si va a sinistra…

Nuovo confronto: 4<6, si va a sinistra…

… 4<5, ancora a sinistra…

… 4>2, si andrebbe a destra ma…

La ricerca termina con un insuccesso.



49

Esempio: Inserimento nodi in un BST

File: **binTreePtrBST.hpp**

```
…
50  bool binTreePtr::BSTsearch(int key) {
51          if(isEmpty())
52                  return false;
53          return doBSTsearch(root,key);
54  }
…
```

50

Esempio: Inserimento nodi in un BST

File: **binTreePtrBST.hpp**

```
31  bool doBSTsearch(binTreePtrNode *n, int key) {
32          int item;
33          item=n->data;
34
35          if(key==item)
36                  return true;
37
38          if(key<item)
39                  if(n->hasLeft())
40                          return doBSTsearch(n->left,key);
41                  else
42                          return false;
43          else
44                  if(n->hasRight())
45                          return doBSTsearch(n->right,key);
46                  else
47                          return false;
48  }
```

51

Alberi binari di ricerca *(Binary Search Tree)*

- Gli algoritmi per
  - determinare il predecessore/successore di un nodo
  - eliminare i nodi

  Sono un po' più complicati…

52

Ordinamento basato su BST

53

Esempio: **BSTSort**

L'algoritmo prende in input un array di *n* elementi non ordinati e restituisce un array con gli stessi elementi disposti in ordine crescente

```
BSTSort(data[n]) {
        myBSTree = new int[n];
        for (i=1;i<=n;i++)
                BSTInsert(myBSTree,data[i]);
        data ← BSTInorderTrav(myBSTree);
}
```

54

Esempio: **`BSTSort`**

Analizziamo le prestazioni dell'algoritmo…

- Complessità di spazio:
  - **`BSTSort`** non ordina *localmente* è necessario almeno un altro array (per l'albero): $O(n) + O(n)$
- Complessità di tempo:
  - L'inserimento di *n* elementi prende $O(n) * T(n)$, dove il secondo è il tempo di inserimento di un solo nodo. Se l'albero è completo o quasi completo, $T(n) = O(h)$ (*h* è l'altezza dell'albero)
  - La ricerca impiega in media $T(n) = O(h)$

55

Gli alberi binari di ricerca *non sono tutti uguali*

Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:

56

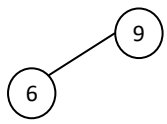
Gli alberi binari di ricerca *non sono tutti uguali*

Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:

57

Gli alberi binari di ricerca *non sono tutti uguali*

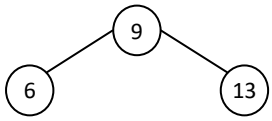
Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:



58

Gli alberi binari di ricerca *non sono tutti uguali*

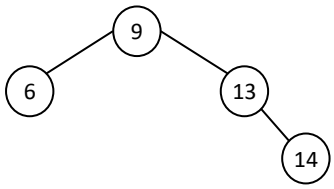
Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:



59

Gli alberi binari di ricerca *non sono tutti uguali*

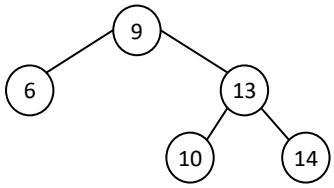
Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:



60

Gli alberi binari di ricerca *non sono tutti uguali*

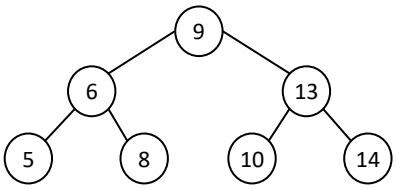
Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:



61

Gli alberi binari di ricerca *non sono tutti uguali*

Costruiamo un BST a partire dalla sequenza 9, 6, 13, 14, 10, 8, 5:



62

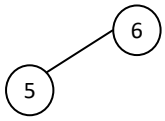
Gli alberi binari di ricerca *non sono tutti uguali*

E ora: costruiamolo a partire dalla sequenza 6, 5, 8, 14, 13, 10, 9:

63

Gli alberi binari di ricerca *non sono tutti uguali*

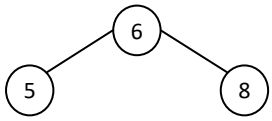
E ora: costruiamolo a partire dalla sequenza 6, 5, 8, 14, 13, 10, 9:



64

Gli alberi binari di ricerca *non sono tutti uguali*

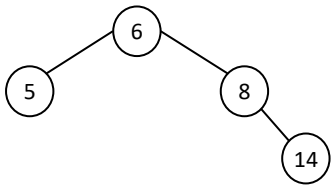
E ora: costruiamolo a partire dalla sequenza 6, 5, 8, 14, 13, 10, 9:



65

Gli alberi binari di ricerca *non sono tutti uguali*

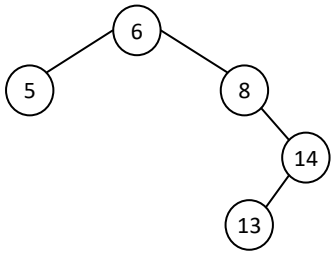
E ora: costruiamolo a partire dalla sequenza 6, 5, 8, 14, 13, 10, 9:



66

Gli alberi binari di ricerca *non sono tutti uguali*

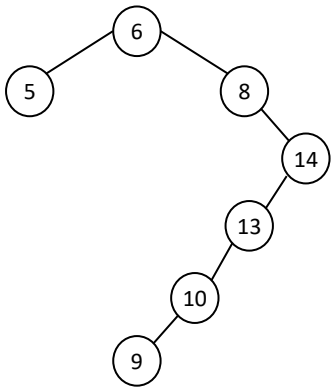
E ora: costruiamolo a partire dalla sequenza 6, 5, 8, 14, 13, 10, 9:



67

Gli alberi binari di ricerca *non sono tutti uguali*

E ora: costruiamolo a partire dalla sequenza 6, 5, 8, 14, 13, 10, 9:



68

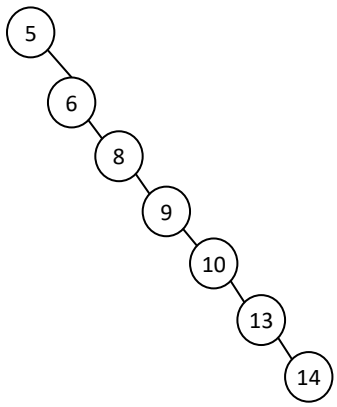
Gli alberi binari di ricerca *non sono tutti uguali*

E ora: costruiamolo a partire dalla sequenza 5, 6, 8, 9, 10, 13, 14:

69

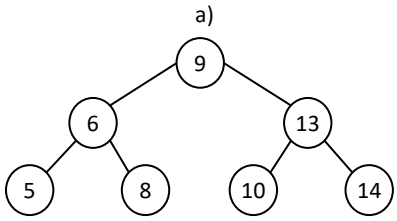
Gli alberi binari di ricerca *non sono tutti uguali*

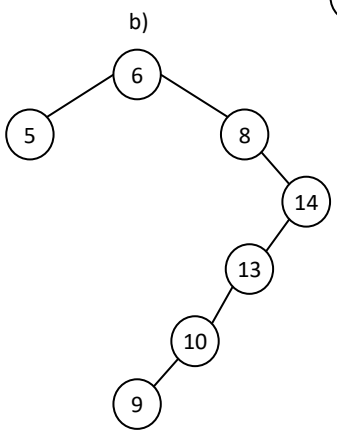
E ora: costruiamolo a partire dalla sequenza 5, 6, 8, 9, 10, 13, 14:

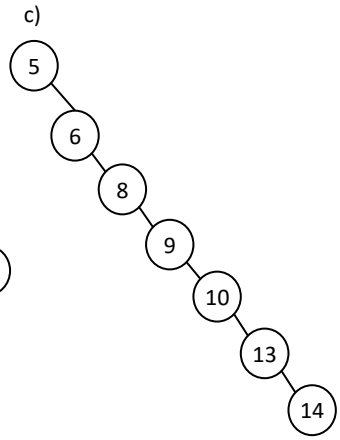


70

Gli alberi binari di ricerca *non sono tutti uguali*







Tempi medi di ricerca e inserimento di un singolo nodo:

a) $O(h)$
b) $O(n)$
c) $O(n)$

71

Prestazioni di **`BSTSort`**

- Valutiamo la complessità di tempo di **`BSTSort`** in presenza di un albero *«equilibrato» (caso medio)*:
  - L'inserimento di *n* elementi prende $n * O(h) = O(n \log n)$
  - Il tempo di ricerca è in $O(h) = O(\log n)$

72

Prestazioni di **`BSTSort`**

- Nel caso *peggiore (worst case)*:
  - L'inserimento di *n* elementi prende $n * O(h) = O(n^2)$
  - Il tempo di ricerca è in $O(n)$

73