Arboles binarios de búsqueda

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Noviembre 2009

Outline

Previously en la clase

- Vimos como usar estructuras de arboles binario para implementar filas de prioridad: permiten establecer una relación de orden parcial en la estructura de datos para recuperar rápido un máximo
- Existen otros tipos de bosques para mejorar el comportamiento en caso de requerir fusiones (montículos binomiales) o de estar interesado por eficiencia amortizada (montículos de Fibonacci)
- Con los arboles binarios de búsqueda, la meta será mas bien buscar si un elemento es presente o no (anuario, diccionario. . .)

Outline

Contenedor set

La filosofía de este contenedor es de almacenar datos de valores únicos, y de poder rápidamente poder determinar una relación de membrecía de un objeto con respeto a este contenedor (noción de conjunto matemático). Idem para Map, MultiSet, MultiMap

Mas generalmente se puede definir un Tipo de Dato Abstracto correspondiente: la tabla de símbolos, que contendrá objetos que se pueden comparar con una relación de orden

Tabla de símbolos

El ADT SymbolTable

- Insert(o): inserta un objeto
- Search(o): busca el objeto o en la estructura
- Remove(o): quita el objeto o de la estructura de datos
- Show(): presenta todos los elementos de la estructura en el orden creciente
- Select(k): selecciona el k-simo elemento
- Merge($otra$): fusiona dos tablas

Tabla de símbolos

Una clase abstracta, suponiendo que los Items tienen un Key:

```
template <class Item, class Key>
class SymbolTable {
    public:
      SymbolTable(int)=0;
      virtual int size()=0;
      virtual void insert(Item)=0;
      virtual Item search(Key)=0;
      virtual void remove(Item)=0;
      virtual void show(std::ostream&)=0;
      virtual Item select(int)=0;
  };
```

Tabla de símbolos

Variantes con varias instancias con la misma llave o no (ex: Set o MultiSet)

- Guardar en las estructuras pasadas una lista de objetos asociados a una llave (facilita search() o remove())
- Dejar la posibilidad de incorporar objetos con llaves múltiples, y el search() de un objeto se hace sobre el primer objeto encontrado con la llave requerida
- Usar un identificador único ademas de la llave (0, 1, 2...)

Tabla de símbolos: llave como índex

En el caso particular en que las llaves son “pequeños” enteros: usar la llave como índex de un arreglo

- operaciones $\mathrm{search}()$, $\mathrm{remove}()$, $\mathrm{insert}()$ en tiempo constante, $\mathrm{select}()$ y $\mathrm{show}()$ en tiempo lineal
- Se puede considerar llaves múltiples con listas enraizadas en cada casilla, por ejemplo
- Pero muy limitado: si las llaves toman valor cualquiera, ya no sirve. . .

Tabla de símbolos: arreglos no ordenados

- Rellenar una con nuevos objetos en donde se pueda (flojo)
- Inserción rápida ($O(1)$)
- operaciones $\mathrm{search}()$, $\mathrm{remove}()$ lineales
- operaciones $\mathrm{select}()$ y $\mathrm{show}()$ en $N \log N$ con los algoritmos clásicos de ordenamiento
- no muy útil si hay muchas invocaciones a $\mathrm{search}()$

Tabla de símbolos: arreglos ordenados

- Mantener en permanencia el orden (no flojo)
- Inserción lenta ($O(N)$)
- operaciones $\text{select}()$ constante y $\text{show}()$ lineal
- uso de la búsqueda binaria para $\text{search}()$: $\log N$
- una mejor opción que considerar... pero penaliza lo de tener inserciones en tiempo lineal

Tabla de símbolos: listas

- Lo mismo que precedentemente, pero con una lista
- Ordenar los elementos o no (flojo/no flojo)
- Comportamiento casi igual, excepto el $\mathrm{select}()$, que es lineal en el caso de una lista ordenada (hay que ir hasta el k-ésimo)

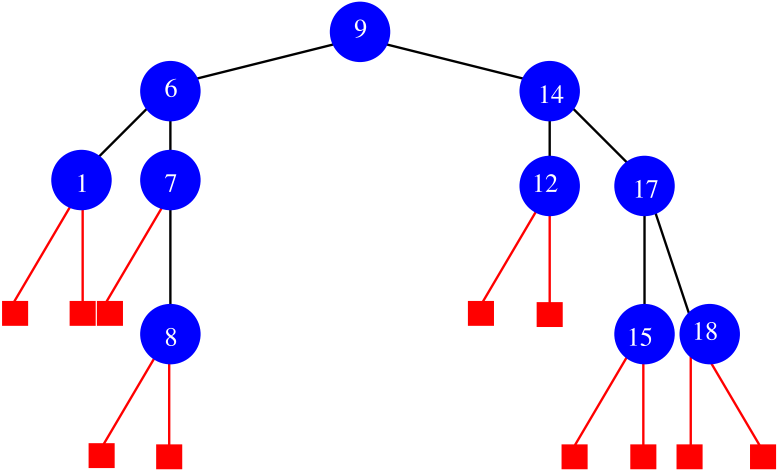
Arboles binarios de búsqueda

Un árbol binario de búsqueda (*Binary Search Tree*) es un árbol binario con llaves asociadas a los nodos y tal que en cada nodo los valores de las llaves de los nodos del sub-árbol izquierda son inferiores a la llave de este nodo, mientras que las del subarbol derecha son superiores. Permite

- inserción, búsqueda rápidas en $O(\log N)$, si el árbol esta equilibrado o aun simplemente "normal"
- show lineal

Outline

Arboles binarios de búsqueda



Arboles binarios de búsqueda

Unas propiedades:

- El máximo esta en el nodo mas a la derecha
- El mínimo esta en el nodo mas a la izquierda
- Recorrer el árbol de manera in-orden nos da el recorrido ordenado de los nodos

Arboles binarios de búsqueda

Una implementación:

```
template <class Item, class Key>
class SymbolTableBST: public SymbolTable<Item,Key>{
 private:
   Item nullItem;
   struct Node {
      Item item; struct Node *l, *r;
      Node(Item x){item = x;l=NULL;r=NULL;}
   };
   Node *head;
   // Search starting from t
   Item searchRecursive(Node *t, Key v);
   void insertRecursive(Node *& h, Item x)
   ...
```

Arboles binarios de búsqueda

Una implementación (seguida):

```
...
public:
 SymbolTableBST(int maxN) :
     SymbolTable<Item,Key>(maxN) { head = NULL;}
  Item search(Key v) {
     return searchRecursive(head, v);
  }
  void insert(Item x) {
     insertRecursive(head, x);
  }
};
```

Arboles binarios de búsqueda

Una implementación (seguida): función recursiva de búsqueda

```
Item SymbolTableBST<Item,Key>::searchRecursive
(Node *h, Key v) {
  if (h == NULL) return nullItem;
  Key t = h->item.key();
  if (v == t) return h->item;
  if (v < t) return searchRecursive(h->l, v);
  else return searchRecursive(h->r, v);
}
void SymbolTableBST<Item,Key>::insertRecursive
(Node*& h, Item x) {
  if (h == NULL) { h = new Node(x); return; }
  if (x.key()<h->item.key())
      insertRecursive(h->l, x);
  else insertRecursive(h->r, x);
 }
```

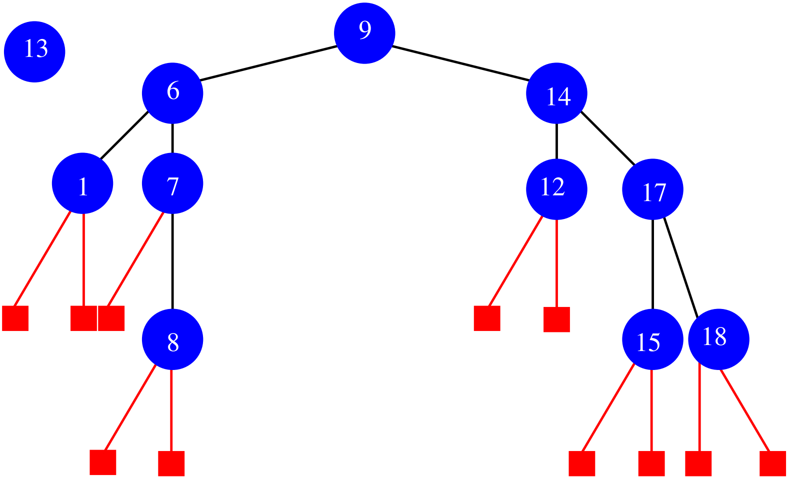
Arboles binarios de búsqueda

Inserción:

- Una primera manera es hacer la inserción en los nodos externos
- "Dejar caer" el nodo hasta llegar a un nodo terminal y reemplazar el nodo terminal por el nuevo nodo
- En cada etapa, comparar el nodo con el nodo corriente y pasar a la izquierda si es $<$, a la derecha sino
- Corresponde al método $\mathrm{insert}()$
- Si el árbol esta equilibrado, complejidad en $O(\log N)$

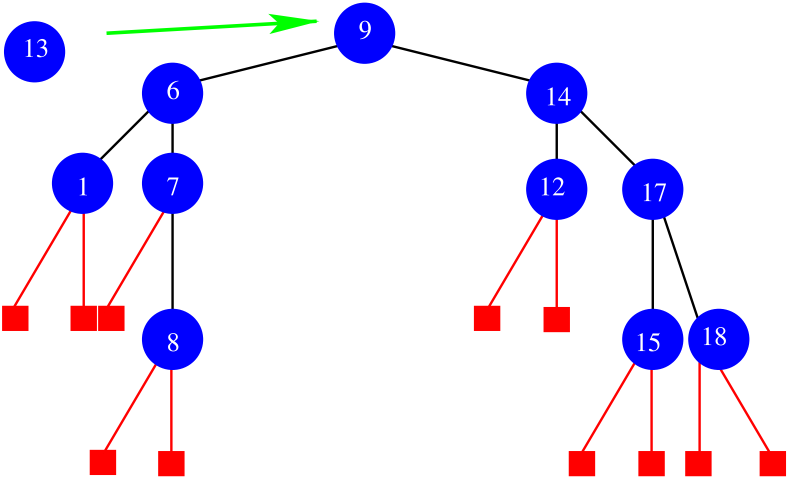
Arboles binarios de búsqueda

Inserción:



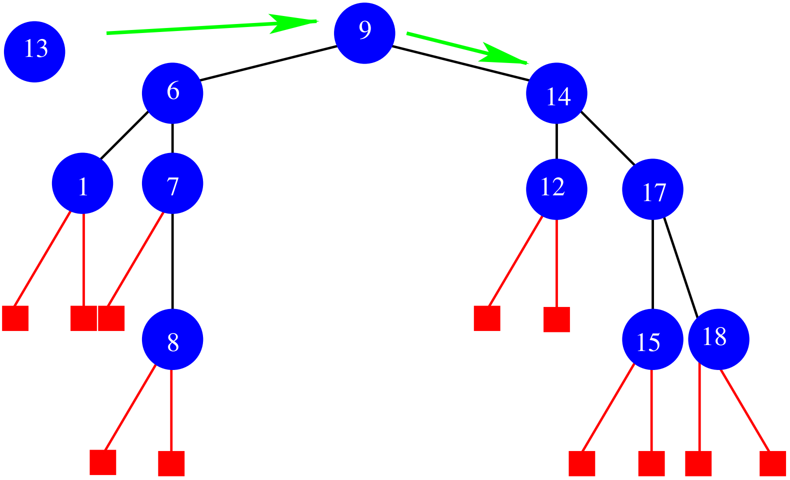
Arboles binarios de búsqueda

Inserción:



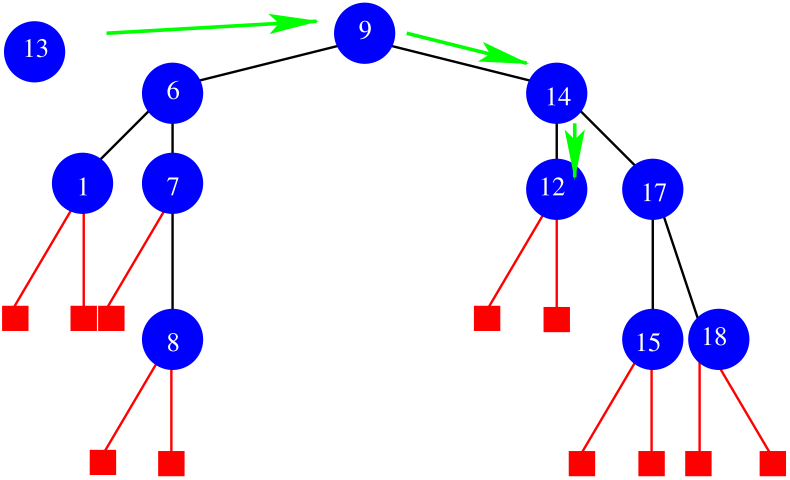
Arboles binarios de búsqueda

Inserción:



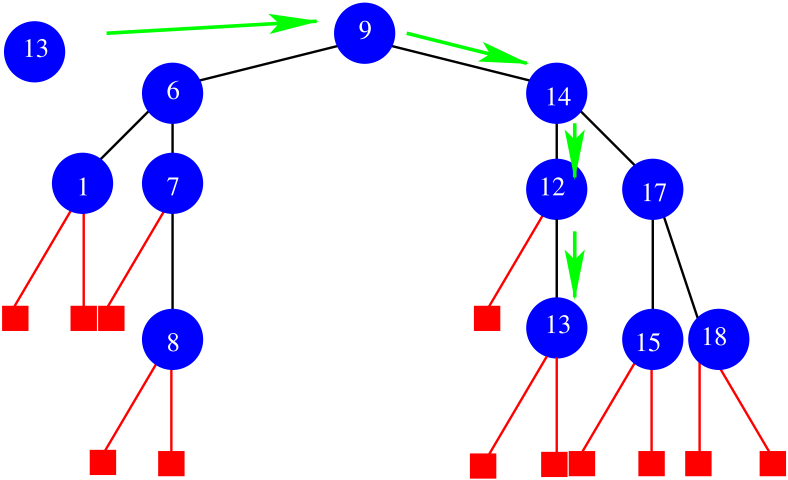
Arboles binarios de búsqueda

Inserción:



Arboles binarios de búsqueda

Inserción:



Arboles binarios de búsqueda

Inserción: duplicados

- por construcción, aparecerán dispersados en la estructura (lo que no facilita las tareas de tipo $\mathrm{searchAll}()\ldots$)
- pero estarán todos en el mismo recorrido de búsqueda (siguiendo la búsqueda aunque ya se encontró nodos de esta llave)

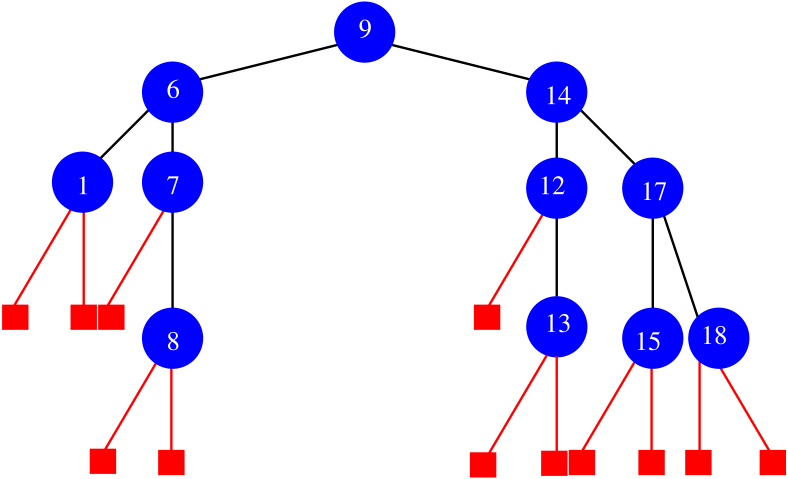
Arboles binarios de búsqueda

Quitar nodo:

- en la versión básica y formal, buscar el nodo que se quiere quitar y reemplazarlo por uno de los nodos de abajo
- dos posibilidades a priori, reemplazar por el mínimo de los mayorantes o por el máximo de los minorantes
- complejidad en $O(\log N)$.

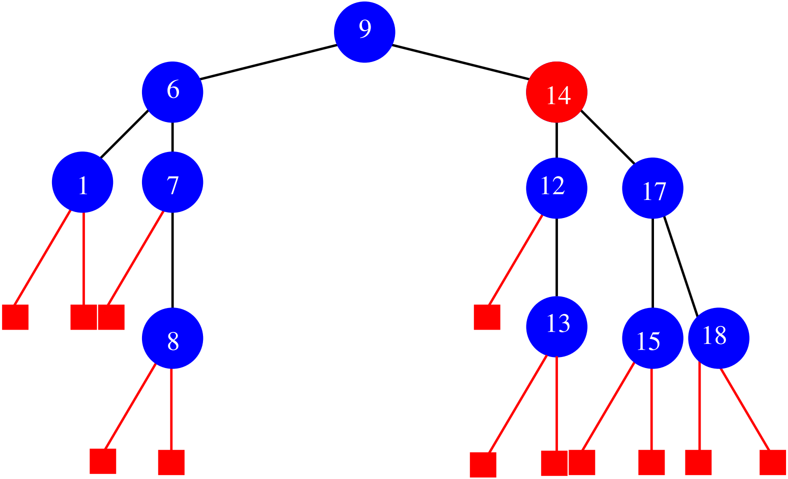
Arboles binarios de búsqueda

Quitar nodo:



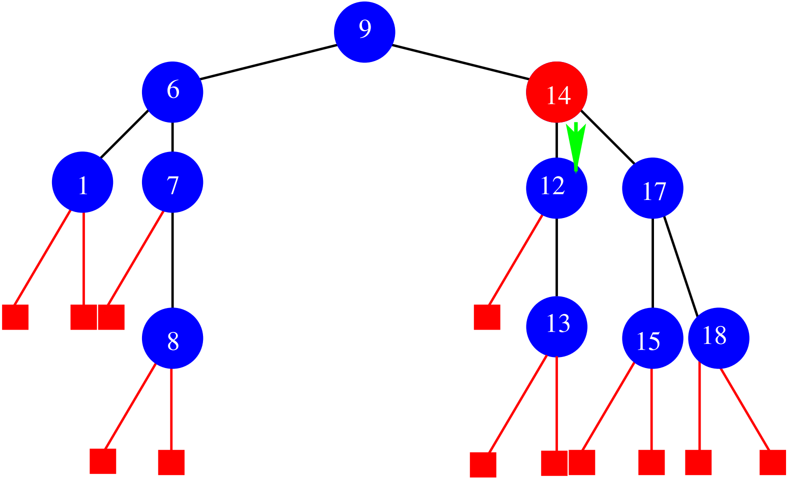
Arboles binarios de búsqueda

Quitar nodo:



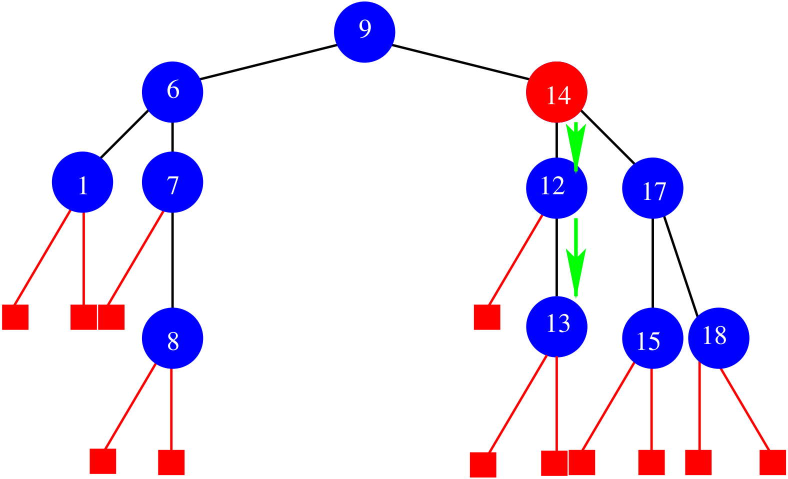
Arboles binarios de búsqueda

Quitar nodo:



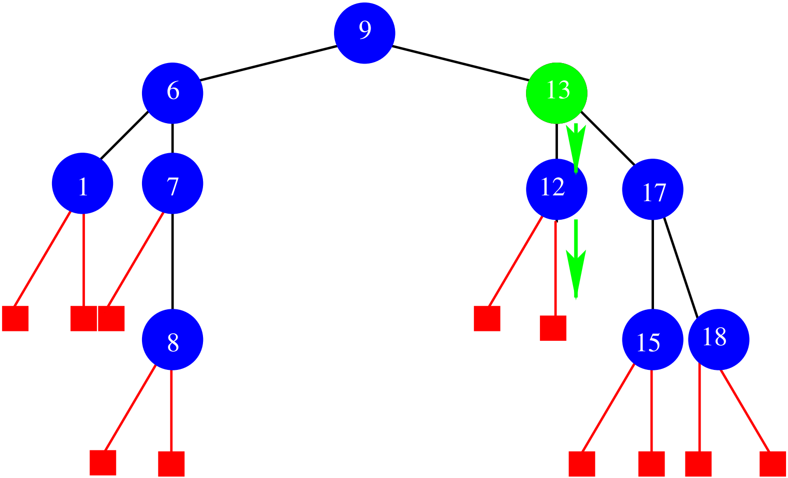
Arboles binarios de búsqueda

Quitar nodo:



Arboles binarios de búsqueda

Quitar nodo:



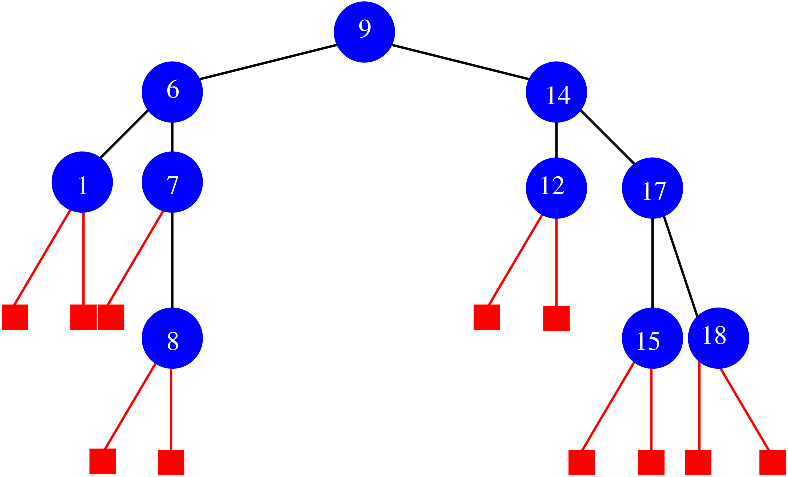
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:

```
private:
 void showRecursive(Node *h, std::ostream& os) {
  if (h == NULL) return;
  // In-order
  showRecursive(h->l, os);
  h->item.show(os);
  showRecursive(h->r, os);
 }
public:
 void show(ostream& os){
  showRecursive(head, os);
 }
```

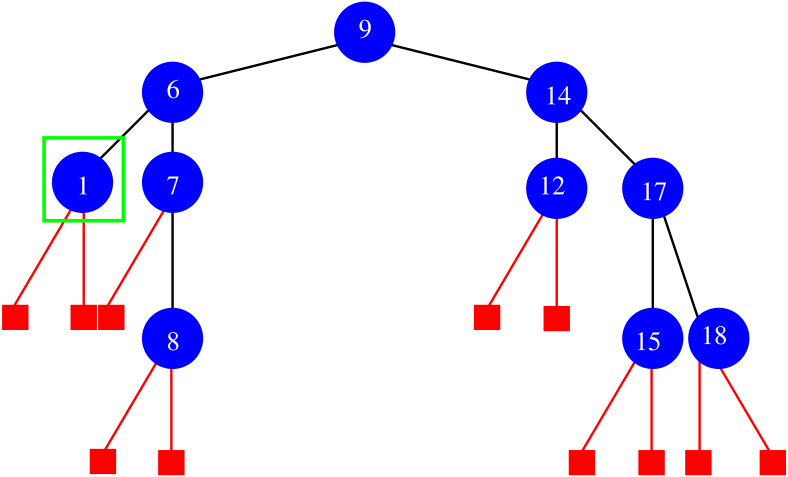
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



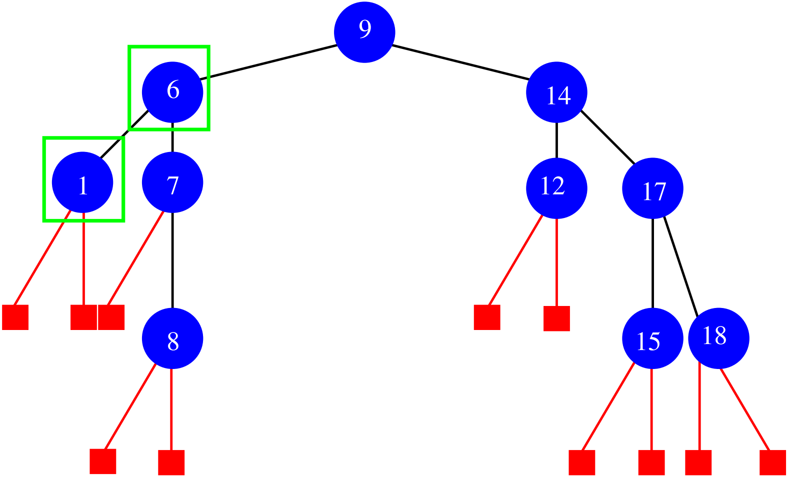
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



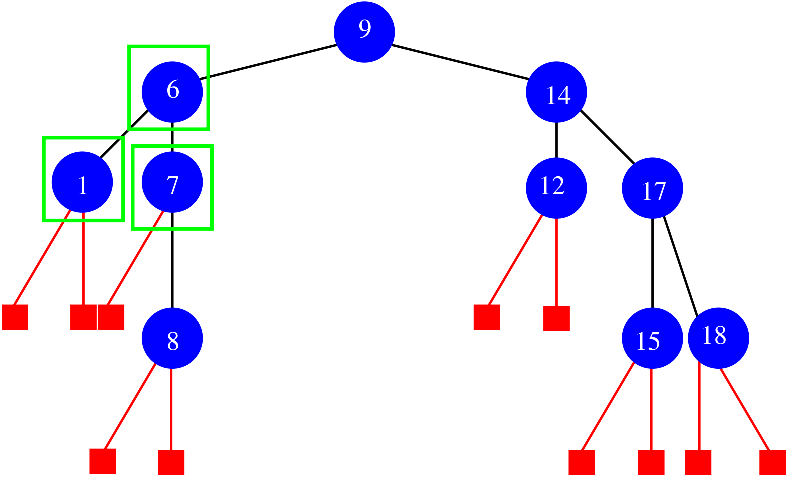
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



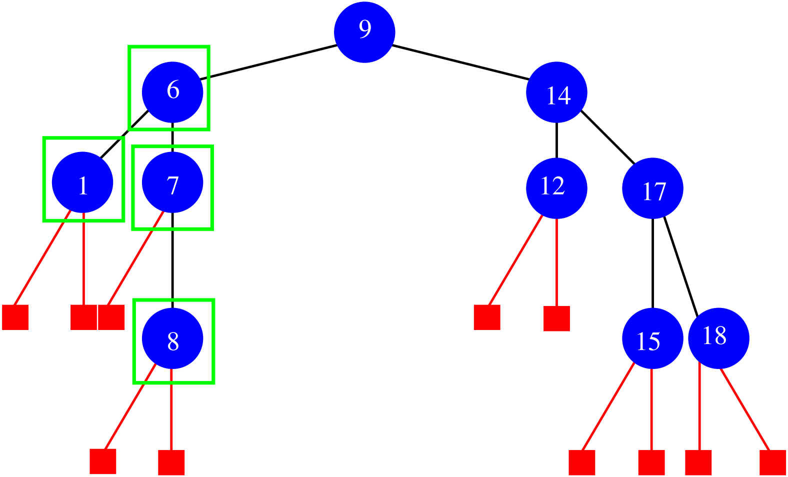
Arboles binarios de búsqueda

Operación de $\mathrm{show}()$ por recorrido in-orden:



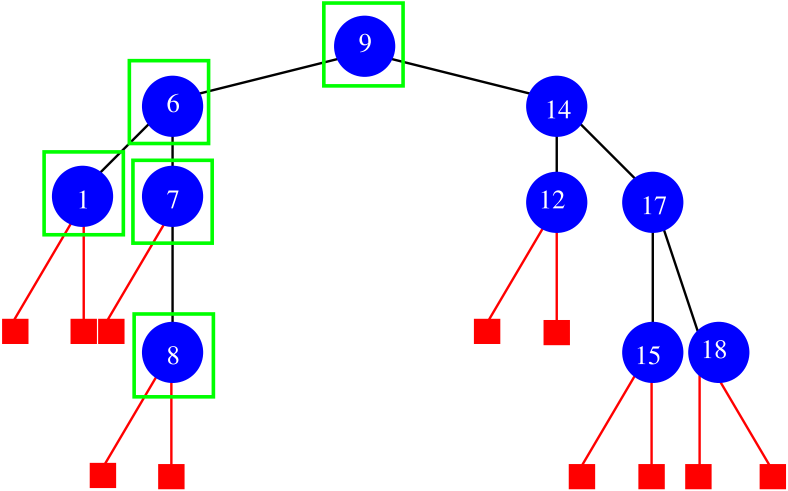
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



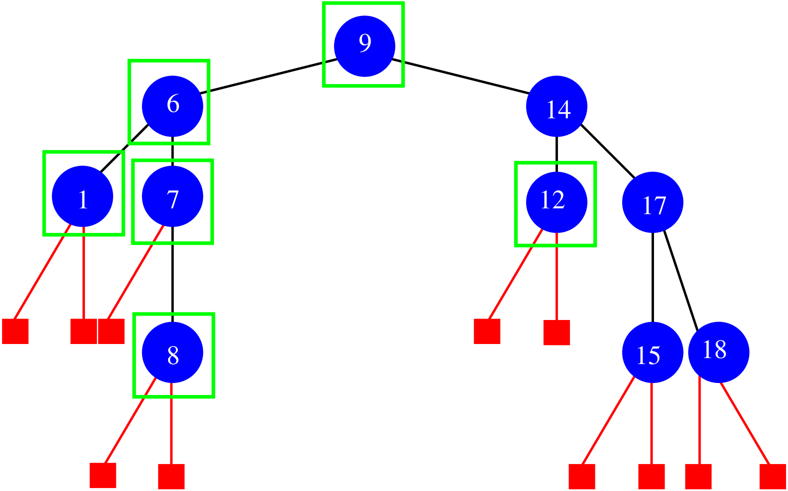
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



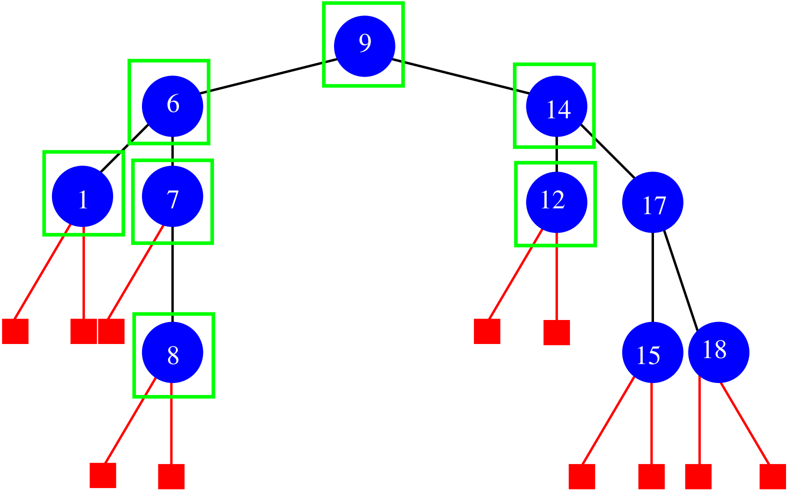
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



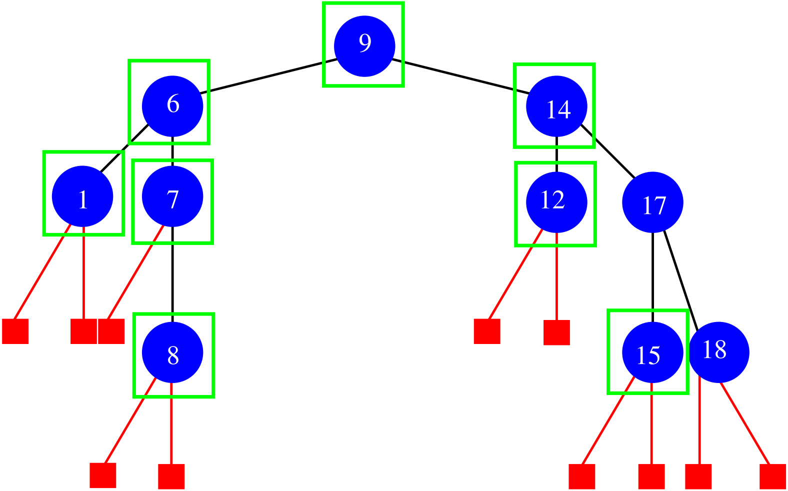
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



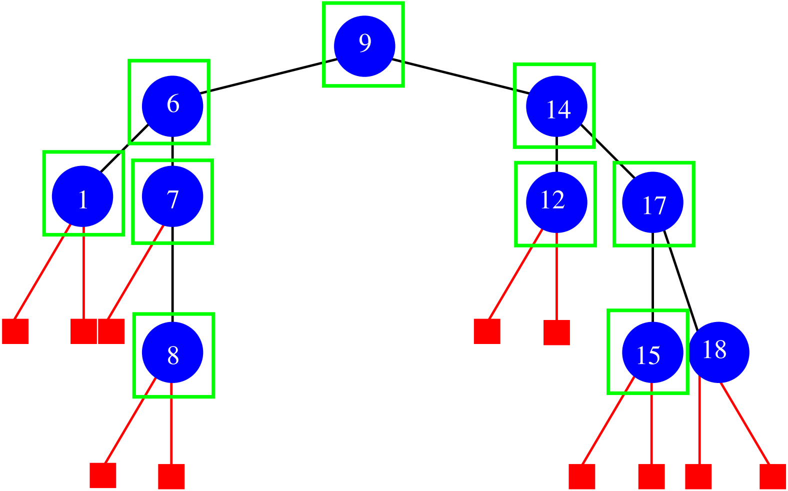
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



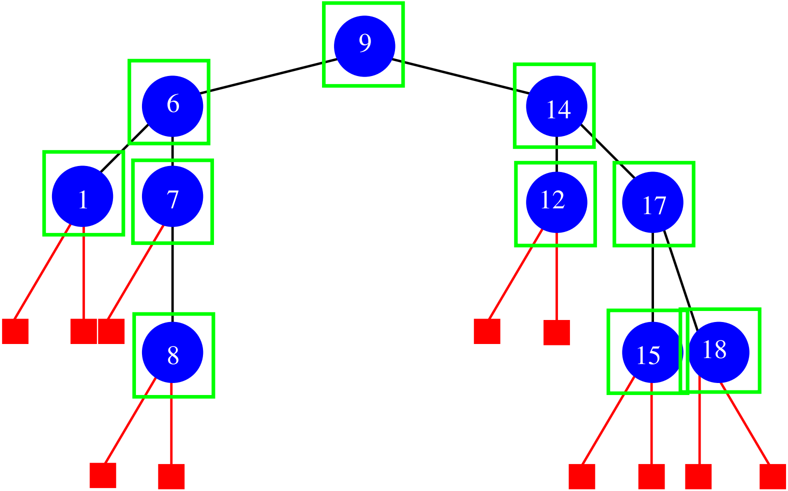
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



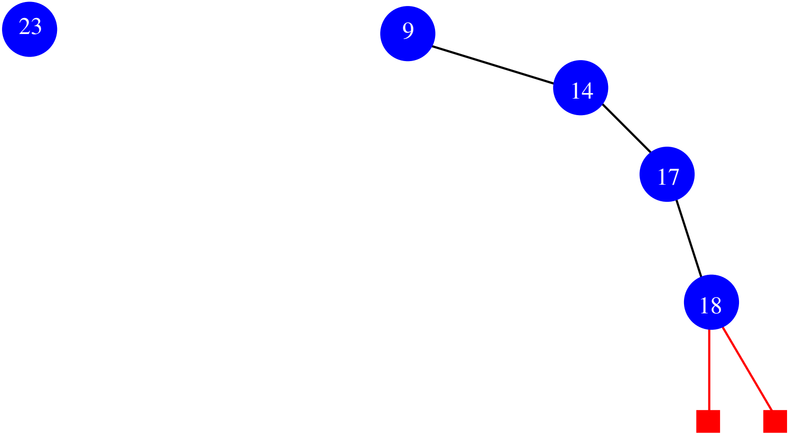
Arboles binarios de búsqueda

Operación de show() por recorrido in-orden:



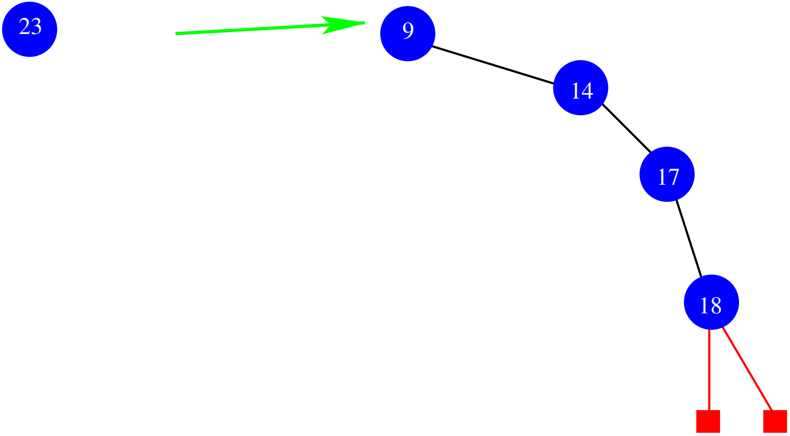
Arboles binarios de búsqueda

Inserción o búsqueda, caso peor:



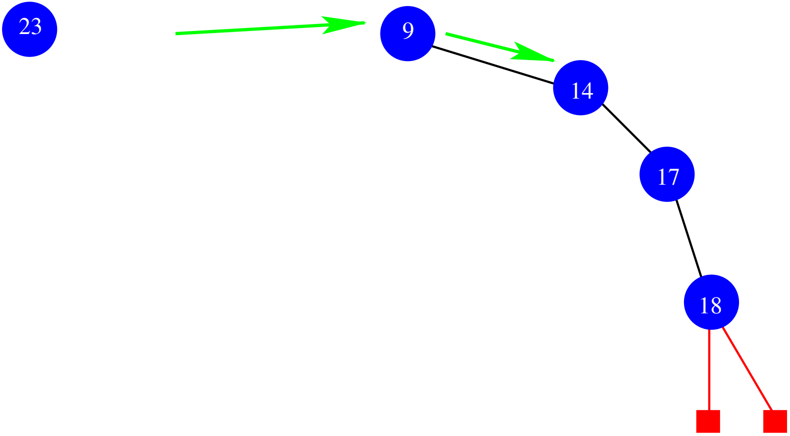
Arboles binarios de búsqueda

Inserción o búsqueda, caso peor:



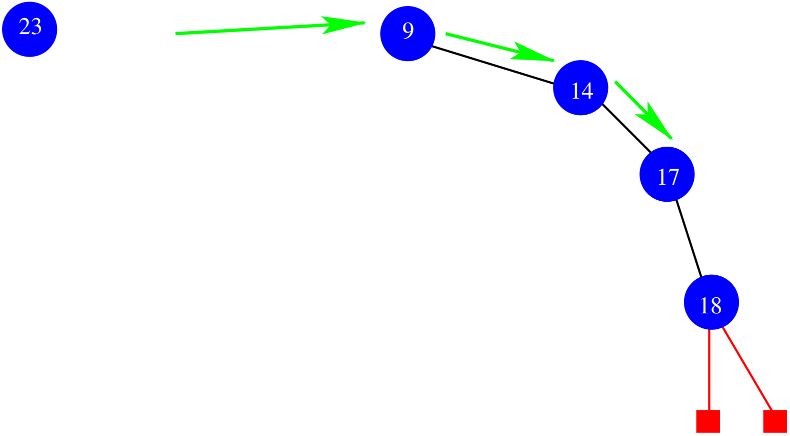
Arboles binarios de búsqueda

Inserción o búsqueda, caso peor:



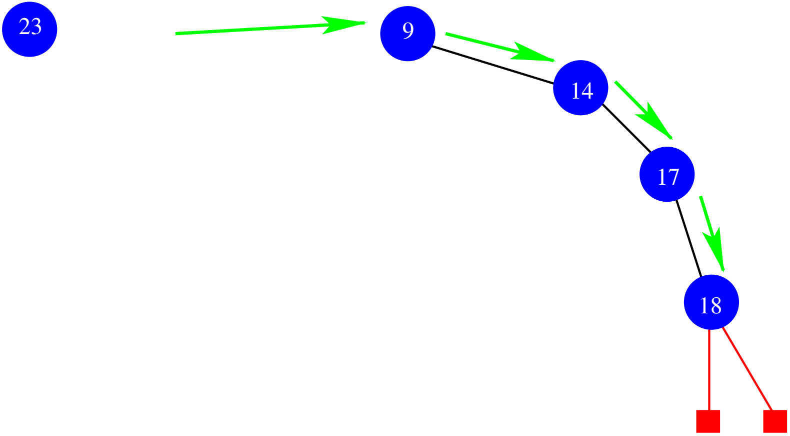
Arboles binarios de búsqueda

Inserción o búsqueda, caso peor:



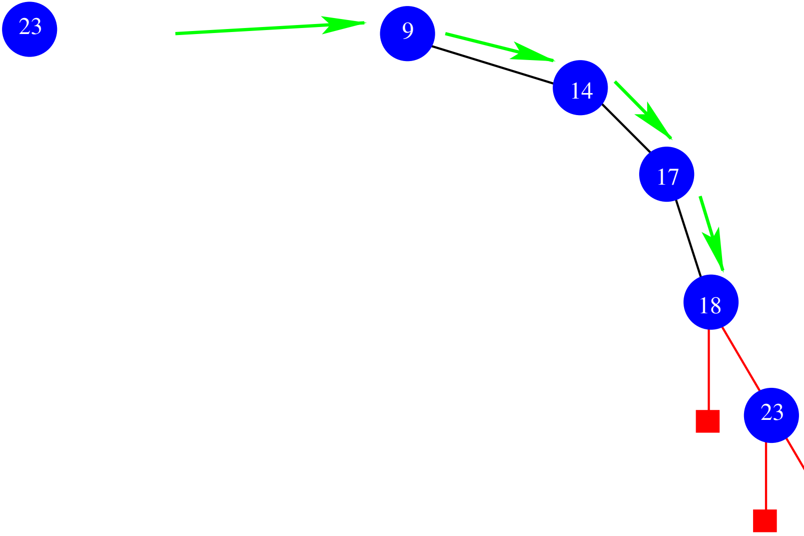
Arboles binarios de búsqueda

Inserción o búsqueda, caso peor:



Arboles binarios de búsqueda

Inserción o búsqueda, caso peor:



Arboles binarios de búsqueda

Búsquedas:

- Caso peor: recorrido en $O(N)$
- Arbol completo o perfectamente equilibrado: a priori $O(\log N)$, pero sería mucha suerte obtener un árbol perfectamente equilibrado a partir de una construcción aleatoria, ¿no?
- Un recuerdo útil, el largo de camino interno se define como la suma de todos los largos de camino entre la raiz y los nodos internos, se nota I_N

Arboles binarios de búsqueda

Búsquedas:

- En el caso genérico de arboles binarios construidos a partir de datos mostrados aleatoriamente: para una búsqueda, el número de operaciones que hacer es 1 plus el largo de camino entre la raíz y este nodo. En promedio sobre este árbol, $1 + \frac{I_N}{N}$
- Ahora, notar que, en promedio sobre todos los arboles binarios de N nodos internos:

$$I_N = N - 1 + \frac{1}{N} \sum_{k=1}^{N} (I_{k-1} + I_{N-k})$$

y que $I_1 = 1$ (¿por qué?)

Arboles binarios de búsqueda

Pero:

$$\begin{aligned} I_N &= N - 1 + \frac{1}{N}\sum_{k=1}^{N}(I_{k-1} + I_{N-k}) \\ &= N - 1 + 2\frac{1}{N}\sum_{k=1}^{N} I_{k-1} \\ NI_N &= N(N-1) + 2\sum_{k=1}^{N} I_{k-1} \end{aligned}$$

Deducimos:

$$\begin{aligned} NI_N - (N-1)I_{N-1} &= N(N-1) - (N-1)(N-2) + 2I_{N-1} \\ NI_N &= 2(N-1) + (N+1)I_{N-1} \\ \frac{I_N}{N+1} &= 2\frac{N-1}{N(N+1)} + \frac{I_{N-1}}{N} \\ \frac{I_N}{N+1} &= 2\frac{1}{N+1} - 2\frac{1}{N(N+1)} + \frac{I_{N-1}}{N} \end{aligned}$$

Arboles binarios de búsqueda

$$
\begin{aligned}
\frac{I_N}{N+1} &= 2\sum_{k=2}^{N}\frac{1}{k+1} - 2\sum_{k=2}^{N}\frac{1}{k(k+1)} + \frac{I_1}{2} \\
&\approx 2\int_1^N \frac{1}{x+1}dx - 2\int_1^N \frac{1}{x(x+1)}dx + \frac{I_1}{2} \\
&\approx 4\int_1^N \frac{1}{x+1}dx - 2\int_1^N \frac{1}{x}dx + \frac{I_1}{2}
\end{aligned}
$$

Entonces:

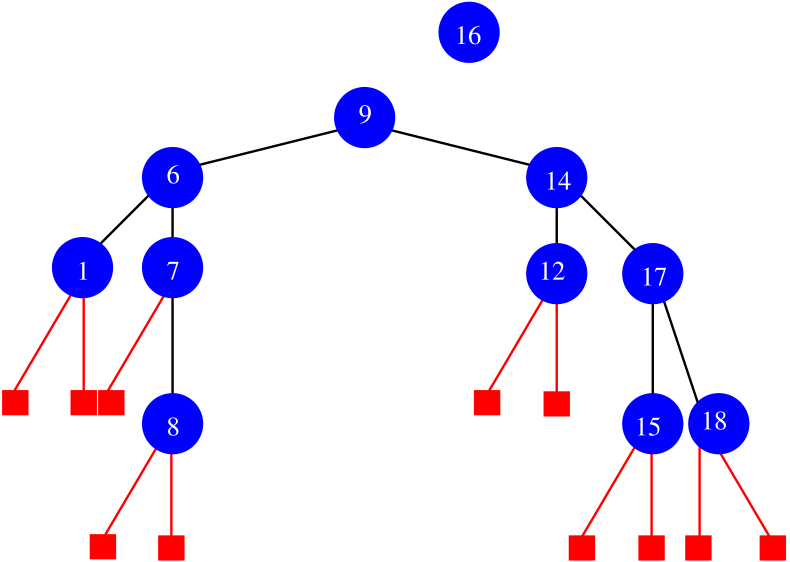
$$\frac{I_N}{N} \approx 2\ln N$$

Inserción en la raíz

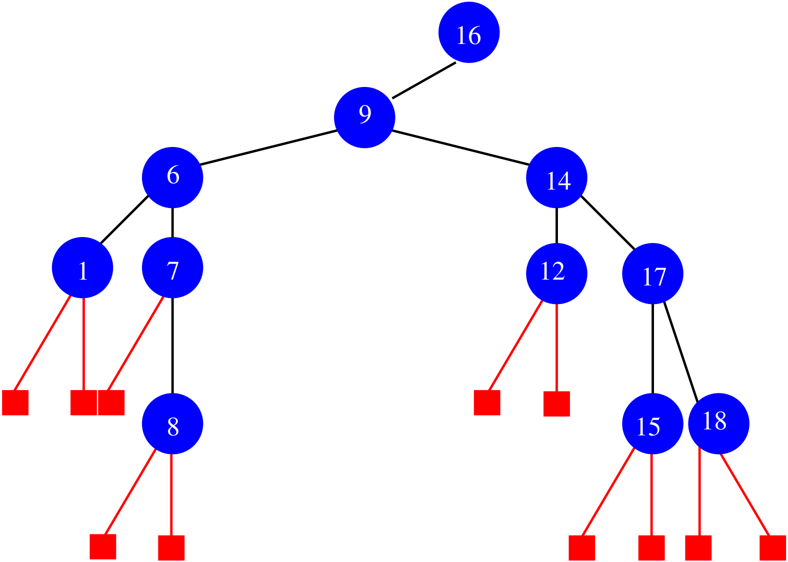
- A priori, no hay razón para hacerlo pero sirve mucho para equilibrar los arboles
- A priori, podríamos intentar ver si la llave del nuevo nodo es superior o no con la raíz actual, y, si sí, formar un nuevo árbol con
  - el nuevo nodo como raíz
  - la previa raíz a la izquierda
  - el sub-árbol de derecha del previo árbol a la derecha

Es suficiente ?

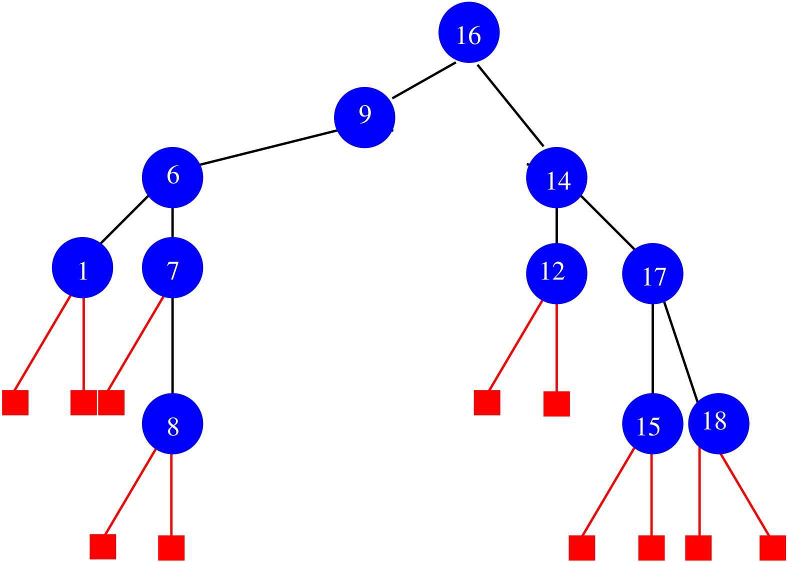
Inserción en la raíz



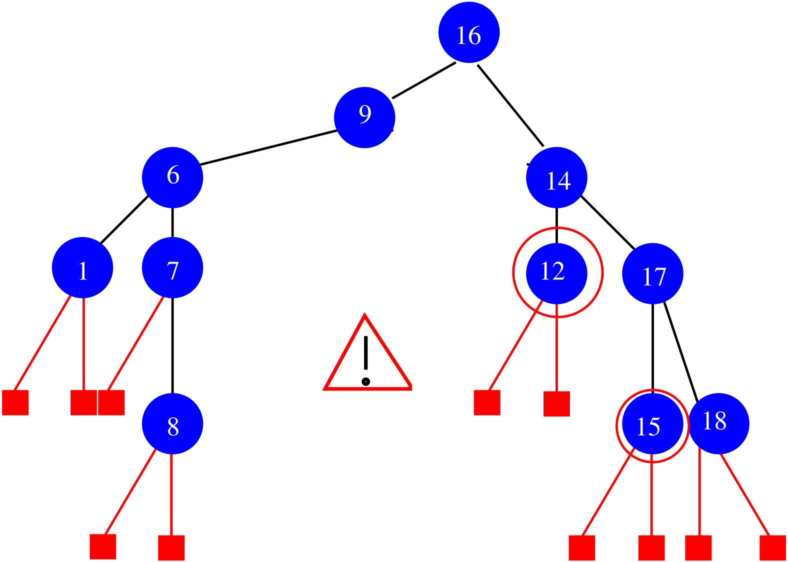
Inserción en la raíz



Inserción en la raíz



Inserción en la raíz

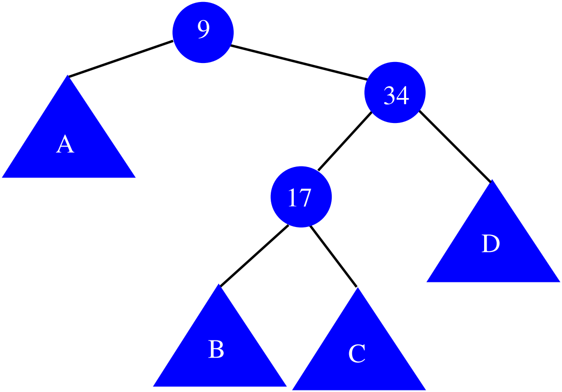


Inserción en la raíz

- De una vez parece difícil…
- Pero, podemos encontrar una manera de hacerlo al revés: empezando con el nodo abajo y haciéndole subir!
- Para eso, necesitamos definir dos operaciones de subida adaptadas a cada uno de los casos: que el nodo que subir sea hijo izquierda o derecha, son las rotaciones

Inserción en la raíz: rotaciones

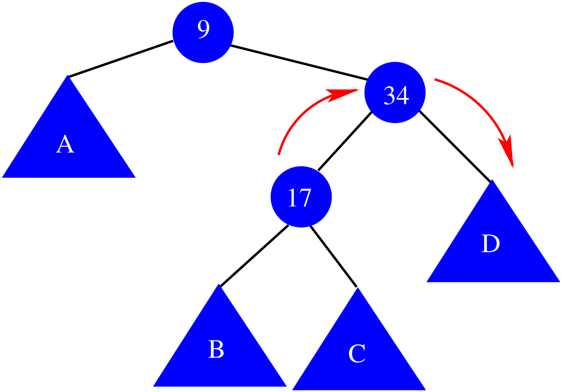
Rotación derecha:



Efecto: sube un nodo hijo izquierda de un nivel hacia la raíz!

Inserción en la raíz: rotaciones

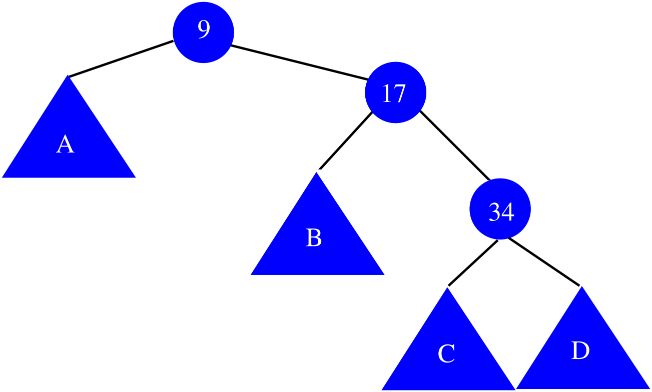
Rotación derecha:



Efecto: sube un nodo hijo izquierda de un nivel hacia la raíz!

Inserción en la raíz: rotaciones

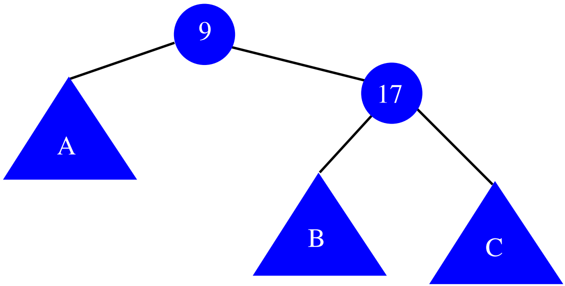
Rotación derecha:



Efecto: sube un nodo hijo izquierda de un nivel hacia la raíz!

Inserción en la raíz: rotaciones

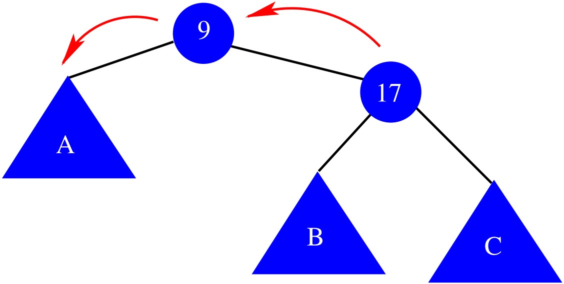
Rotación izquierda:



Efecto: sube un nodo hijo derecha de un nivel hacia la raíz!

Inserción en la raíz: rotaciones

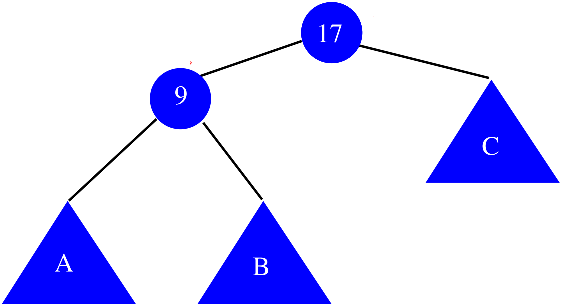
Rotación izquierda:



Efecto: sube un nodo hijo derecha de un nivel hacia la raíz!

Inserción en la raíz: rotaciones

Rotación izquierda:



Efecto: sube un nodo hijo derecha de un nivel hacia la raíz!

Arboles binarios de búsqueda

Implementación:

```
void rotRight(Node *& h) {
 Node *x = h->l;
 h->l = x->r;
 x->r = h;
 h = x;
}
void rotLeft(Node *& h) {
 Node *x = h->r;
 h->r = x->l;
 x->l = h;
 h = x;
 }
```

Sólo tres ligas modificadas, y la estructura queda la de un ABB!

Arboles binarios de búsqueda

Ahora podemos proponer una inserción recursiva en la raíz:

```
private:
 void insertAtRoot(Node *& h, Item x) {
  if (h == NULL) { h = new Node(x); return; }
  if (x.key()<h->item.key())  {
     insertAtRoot(h->l,x);
     rotRight(h);
  } else {
     insertAtRoot(h->r, x);
     rotLeft(h);
  }
 }
public:
 void insert(Item item) {
  insertAtRoot(head, item);
 }
```

Arboles binarios de búsqueda

Ejemplo: hacer la inserción de 5, 7, 1, 3, 8, 10, 4

- Ventaja de tal método: las últimas llaves introducidas están arriba
- Interesante si por ejemplo el mismo proceso esta aplicado en cada $\mathrm{search}()$: así los nodos mas buscados se quedan arriba!
- Ahora qué tal de $\mathrm{select}()$, $\mathrm{merge}()$?

Arboles binarios de búsqueda

select(), con contador en cada nodo:

```
private:
  Item selectRecursive(Node *h, int k) {
    if (h == NULL) return nullItem;
    int t = (h->l == NULL) ? 0: h->l->N;
    if (t > k-1)
       return selectRecursive(h->l, k);
    if (t < k-1)
       return selectRecursive(h->r, k-t-1);
    return h->item;
  }
public:
  Item select(int k)
    { return selectRecursive(head, k); }
```

Arboles binarios de búsqueda

select(), modificado que provoca la subida recursiva del k-ésimo elemento partiendo el árbol en dos sub-arboles de $k-1$ y $N-k$ elementos:

```
void partRecursive(Node *& h, int k) {
 int t = (h->l == NULL) ? 0: h->l->N;
 if (t> k-1) {
  partRecursive(h->l, k); rotRight(h);
 }
 if (t<k-1) {
  partRecursive(h->r, k-t-1); rotLeft(h);
 }
}
```

Es la operación básica para equilibrar arboles...

Arboles binarios de búsqueda

$\text{remove}()$: se puede implementar dada esa función de partición:

- recursivamente quitar el nodo del sub-árbol en que está
- cuando este nodo es raíz del árbol examinado, considerar los dos sub-arboles
- en el sub-arbol de derecha subir el mínimo hasta la raíz, tiene una hoja como hijo izquierda
- reunir el sub-arbol de izquierda

Arboles binarios de búsqueda

remove()

```
private:
 Node *joinLeftRight(Node *a,Node *b) {
  if (b == NULL) return a;
  partRecursive(b,1); b->l = a;
  return b;
 }
 void removeRecursive(Node *& h, Key v) {
  if (h == NULL) return;
  Key w = h->item.key();
  if (v < w) removeRecursive(h->l, v);
  if (w < v) removeRecursive(h->r, v);
  if (v == w) {
     Node *t = h;
     h = joinLeftRight(h->l, h->r); delete t;
    }
```

Arboles binarios de búsqueda

remove()

```
public:
 void remove(Item x) {
  removeRecursive(head, x.key());
 }
```

Un problema: deja el árbol no tan equilibrado (porque introduce un sesgo!)

Arboles binarios de búsqueda

merge() recursivamente, quitando la raiz de uno de los dos y introduciéndola en el otro en la raíz da dos sub-problemas de tipo merge()

```
private:
 Node *mergeRecursive(Node *a,Node *b) {
   if (b == NULL) return a;
   if (a == NULL) return b;
   insertAtRoot(b, a->item);
   b->l = mergeRecursive(a->l, b->l);
   b->r = mergeRecursive(a->r, b->r);
   delete a; return b;
 }
public:
 void merge(SymbolTableBST<Item, Key>& b)
        { head = mergeRecursive(head, b.head); }
```

Arboles equilibrados

- Arboles que garantizan el equilibrio de los arboles
- Equilibrio: la diferencia de altura entre los hijos es siempre al máximo de 1
- Varias técnicas... Arboles (2,4) y rojo-negro