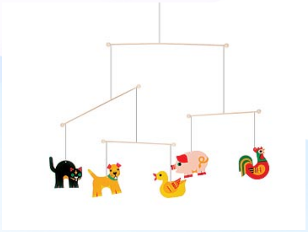
Lecture Notes on Data Structures and Algorithms: Dictionaries

Conrado Martínez
U. Politècnica Catalunya

April 1, 2016



Part III

Dictionaries

Introduction

A dictionary (a.k.a. *associative table*, *map* in C++ parlance, *symbol table*) is a data structure that stores a finite set of elements, each one endowed with unique identifier or key, and that supports search by key.

For instance, a dictionary could store information about the students in a University school and allow to access the information about some student given his/her DNI number or his/her full name, depending on what was choosen to act as a *key*; the full name is probably not a good choice because, although unlikely, two students might have exactly the same full name.

Introduction

There are many different mathematical models, all equivalent, for dictionaries:

1. A dictionary D is a partial function with finite domain[1] from the set of keys K to a set of values V.
2. A dictionary is a total function from the set of keys K to the set $V \cup \{\perp\}$, where V is a set of values, $\perp \notin V$ is a special value and
$$\{k \in K \mid D(k) \in V\}$$
is finite.
3. A dictionary D is a finite set of pairs $\langle k, v\rangle \in K \times V$, where K is the set of keys, V is the set of values, and no two pairs in the dictionary have the same key.
4. A dictionary D is a finite set of elements from a domain $\mathcal{U}$, where each element x has a key ($\text{KEY} : \mathcal{U} \longrightarrow K$), for any two different elements x and y in D, we have $\text{KEY}(x) \neq \text{KEY}(y)$.

[1] The subset of values $k \in K$ for which the function has an image is finite.

Introduction

There are many different mathematical models, all equivalent, for dictionaries:

1. A dictionary D is a partial function with finite domain[1] from the set of keys K to a set of values V.
2. A dictionary is a total function from the set of keys K to the set $V \cup \{\perp\}$, where V is a set of values, $\perp \notin V$ is a special value and
$$\{k \in K \mid D(k) \in V\}$$
is finite.
3. A dictionary D is a finite set of pairs $\langle k, v\rangle \in K \times V$, where K is the set of keys, V is the set of values, and no two pairs in the dictionary have the same key.
4. A dictionary D is a finite set of elements from a domain $\mathcal{U}$, where each element x has a key ($\textsc{key} : \mathcal{U} \longrightarrow K$), for any two different elements x and y in D, we have $\textsc{key}(x) \neq \textsc{key}(y)$.

[1] The subset of values $k \in K$ for which the function has an image is finite.

Introduction

There are many different mathematical models, all equivalent, for dictionaries:

1. A dictionary D is a partial function with finite domain[1] from the set of keys K to a set of values V.
2. A dictionary is a total function from the set of keys K to the set $V \cup \{\perp\}$, where V is a set of values, $\perp \notin V$ is a special value and
$$\{k \in K \mid D(k) \in V\}$$
is finite.
3. A dictionary D is a finite set of pairs $\langle k, v\rangle \in K \times V$, where K is the set of keys, V is the set of values, and no two pairs in the dictionary have the same key.
4. A dictionary D is a finite set of elements from a domain $\mathcal{U}$, where each element x has a key ($\text{KEY} : \mathcal{U} \longrightarrow K$), for any two different elements x and y in D, we have $\text{KEY}(x) \neq \text{KEY}(y)$.

[1]The subset of values $k \in K$ for which the function has an image is finite.

Introduction

There are many different mathematical models, all equivalent, for dictionaries:

1. A dictionary D is a partial function with finite domain[1] from the set of keys K to a set of values V.
2. A dictionary is a total function from the set of keys K to the set $V \cup \{\perp\}$, where V is a set of values, $\perp \notin V$ is a special value and
$$\{k \in K \mid D(k) \in V\}$$
is finite.
3. A dictionary D is a finite set of pairs $\langle k, v \rangle \in K \times V$, where K is the set of keys, V is the set of values, and no two pairs in the dictionary have the same key.
4. A dictionary D is a finite set of elements from a domain $\mathcal{U}$, where each element x has a key ($\textsf{KEY} : \mathcal{U} \longrightarrow K$), for any two different elements x and y in D, we have $\textsf{KEY}(x) \neq \textsf{KEY}(y)$.

[1] The subset of values $k \in K$ for which the function has an image is finite.

Introduction

A particular important instance of dictionaries are finite sets (in the traditional mathematical sense). It is enough to see the finite set as a subset of K and the dictionary as the characteristic function of K, with $V = \{\textbf{true}\}$ and $\perp = \textbf{false}$. Or use the last definition, taking $\text{KEY}(x) = x$.
Search by key in a set S is hence just a Boolean function to determine membership, i.e., does x belong or not to the set S.

Introduction

According to the operations that an Abstract Data Type (ADT) dictionary supports —search by key is always provided— we can classify them into several categories:

- Static dictionaries: the constructor is given the list of n elements to store in the dictionary, no further insertions or deletions are permitted.
- Semi-dynamic dictionaries: new pairs $\langle k, v \rangle$ can be added to the dictionary (updating the associated value to a given key is also permitted), but no pair can be removed from the dictionary. It is common that the class offers a constructor to create an empty dictionary.
- (Fully) dynamic dictionaries: insertions and deletions (given the key) are both supported.

Introduction

According to the operations that an Abstract Data Type (ADT) dictionary supports —search by key is always provided— we can classify them into several categories:

- Static dictionaries: the constructor is given the list of n elements to store in the dictionary, no further insertions or deletions are permitted.
- Semi-dynamic dictionaries: new pairs $\langle k, v \rangle$ can be added to the dictionary (updating the associated value to a given key is also permitted), but no pair can be removed from the dictionary. It is common that the class offers a constructor to create an empty dictionary.
- (Fully) dynamic dictionaries: insertions and deletions (given the key) are both supported.

Introduction

According to the operations that an Abstract Data Type (ADT) dictionary supports —search by key is always provided— we can classify them into several categories:

- Static dictionaries: the constructor is given the list of n elements to store in the dictionary, no further insertions or deletions are permitted.
- Semi-dynamic dictionaries: new pairs $\langle k, v \rangle$ can be added to the dictionary (updating the associated value to a given key is also permitted), but no pair can be removed from the dictionary. It is common that the class offers a constructor to create an empty dictionary.
- (Fully) dynamic dictionaries: insertions and deletions (given the key) are both supported.

Introduction

```
template <typename Key, typename Value>
class Dictionary {
public:
  Dictionaty();
  ~Dictionary();
  ...
  void lookup(const Key& k,
       bool& exists, Value& v) const;
  void insert(const Key& k,
       const Value& v);
  void remove(const Key& k);
  ...
private:
  ...
}
```

Introduction

A subfamily of dictionaries of particular interest are those called sorted, supporting sequential traversal of the elements in increasing order of their keys. It is assumed that the set of keys has a well-defined total order (that is, for any two given keys k and k', we have $k < k'$, $k = k'$ or $k' < k$).

The dictionary class can then provide methods to iterate through all the elements in the dictionary in ascending order of their keys. In C++ dictionary classes, this is usually accomplished through *iterators*, e.g.,

```
// prints the chemical elements in alphabetic order together with their
// atomic weight
// an iterator it into a Dictionary<Key,Value> points to a pair<Key,Value>
Dictionary<string, double> PeriodicTable;
...
for (auto it : PeriodicTable) {
    cout << it->first << ": " << it->second << endl;
}
```

Introduction

It is very common that a dictionary class supports additional operations, besides search by key and insertions/deletions:

- Set-theoretic operations: unions, intersections, set difference, . . .
- Search and removal by rank: find/remove the element with the i-th smallest key, remove the element with the i-th smallest key, . . .
- Search and removal in ranges: find/remove all elements which key is within the range $[k_1..k_2]$, given the keys k_1 and k_2
- Counting: determine how many elements have a key less/greater than a given key k, determine how many elements have a key within the range $[k_1..k_2]$, . . .

Introduction

It is very common that a dictionary class supports additional operations, besides search by key and insertions/deletions:

- Set-theoretic operations: unions, intersections, set difference, ...
- Search and removal by rank: find/remove the element with the i-th smallest key, remove the element with the i-th smallest key, ...
- Search and removal in ranges: find/remove all elements which key is within the range $[k_1..k_2]$, given the keys k_1 and k_2
- Counting: determine how many elements have a key less/greater than a given key k, determine how many elements have a key within the range $[k_1..k_2]$, ...

Introduction

It is very common that a dictionary class supports additional operations, besides search by key and insertions/deletions:

- Set-theoretic operations: unions, intersections, set difference, ...
- Search and removal by rank: find/remove the element with the i-th smallest key, remove the element with the i-th smallest key, ...
- Search and removal in ranges: find/remove all elements which key is within the range $[k_1..k_2]$, given the keys k_1 and k_2
- Counting: determine how many elements have a key less/greater than a given key k, determine how many elements have a key within the range $[k_1..k_2]$, ...

Introduction

It is very common that a dictionary class supports additional operations, besides search by key and insertions/deletions:

- Set-theoretic operations: unions, intersections, set difference, . . .
- Search and removal by rank: find/remove the element with the i-th smallest key, remove the element with the i-th smallest key, . . .
- Search and removal in ranges: find/remove all elements which key is within the range $[k_1..k_2]$, given the keys k_1 and k_2
- Counting: determine how many elements have a key less/greater than a given key k, determine how many elements have a key within the range $[k_1..k_2]$, . . .

Introduction

Under special circumstances, for instance to represent a finite set K such that $K \subset \{0, \ldots, M-1\}$ for some relatively small value M (the number of elements in the set is not $\ll M$ or M is not too large), we can implement K as a bitvector. Search ($\equiv$ membership), insertions and deletions can be done in $\Theta(1)$ time. Bitvectors also efficiently support unions (bit-wise OR), intersections (bit-wise AND), etc.

Introduction

An unsorted linked list can be used to implement a dictionary, with each element of the list storing a pair $\langle key, value\rangle$. The cost of searches, insertions and deletions is $\Theta(n)$, where n is the size of the dictionary. It can be acceptable if n is small, e.g., $n \leq 250$. All algorithms are very simple. Unsorted linked lists also support self-organizing heuristics, e.g. *move-to-front*; this is specially useful when there is a lot of locality of reference in the access pattern.
Unsorted linked lists should not be used if we need to access the dictionary in sorted order.

Introduction

- For static dictionaries and dictionaries where updates (insertions and deletions) occur very unfrequently it might be worth to maintain the dictionary in a sorted array (vector). It is a lightweight solution, with minimal memory requirements. The algorithms are simple and searches are efficient (use binary search for lookups); ordered traversal is also trivial.
- Sorted linked lists combine some of the advantages of (unsorted) linked lists and sorted vectors, although they also inherit some of their inconvenients. They support well the traversal of the dictionary in order, and set-theoretic operations (e.g., unions). Search and update operations are $\Theta(n)$ both in the worst- and the average-case.

Introduction

Some types of keys (e.g., `string`) allow the use of special data structures to implement the dictionary. The most important one is the so-called trie and its many variants.

Dictionaries for strings often support specific operations such as finding all strings in the dictionary starting with a particular prefix or all strings in the dictionary that match a given *regular expression*.

Introduction

- What’s next?
  - Search trees
  - Hash tables
- In later courses, other advanced data structures for dictionaries
  - Skip lists
  - Tries
  - . . . and more

Part III

Dictionaries

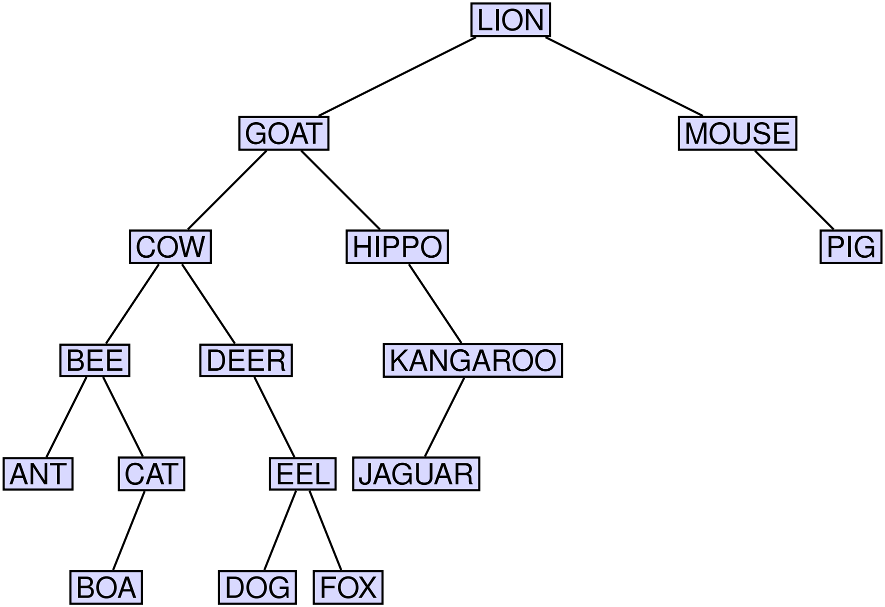
Definition

Definition

A binary search tree (BST, for short) T is a binary tree that is either empty, or it contains at least one element x at its root, and

1. The left and right subtrees of T, L and R, respectively, are binary search trees.
2. For all elements y in L, $\text{KEY}(y) < \text{KEY}(x)$, and for all elements z in R, $\text{KEY}(z) > \text{KEY}(x)$.

An Example


LION
GOAT
MOUSE
COW
HIPPO
PIG
BEE
DEER
KANGAROO
ANT
CAT
EEL
JAGUAR
BOA
DOG
FOX

Binary Search Trees: Basic Operations

```
template <typename Key, typename Value>
class Dictionary {
public:
  ...
  void lookup(const Key& k,
       bool& exists, Value& v) const;
  void insert(const Key& k,
       const Value& v);
  void remove(const Key& k);
  ...
private:
  struct node_bst {
     Key _k;
     Value _v;
     node_bst* _left;
     node_bst* _right;
     // constructor for the class/struct node_bst
     node_bst(const Key& k, const Value& v,
              node_bst* left = nullptr, node_bst* right = nullptr);
  };
  node_bst* root;

  static node_bst* bst_lookup(node_bst* p,
    const Key& k);
  static node_bst* bst_insert(node_bst* p,
    const Key& k, const Value& v);
  static node_bst* bst_remove(node_bst* p,
    const Key& k);
  static node_bst* join(node_bst* t1,
    node_bst* t2);
  static node_bst* relocate_max(node_bst* p);
  ...
}
```

Traversals

Lemma

An inorder traversal of a binary search tree visits all the elements it contains in ascending order of their keys.

Traversals are often implemented via iterators; then the class offers methods like `begin` and `end` (among others) for the traversals.

Traversals

A simpler (but less flexible) strategy is to have a method that “dumps” the dictionary contents into a sorted list.

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
void dump(list<pair<Key,Value>>& L) const {
     L.clear();
     bst_dump(root, L);
}
...
private:
...
static void bst_dump(node_bst* p, list<pair<Key,Value>>& L) {
    if (p != nullptr) {
       bst_dump(p -> _left, L);
       L.push_back(make_pair(p -> _k, p -> _v));
       bst_dump(p -> _right, L);
    }
}
...
```

Search

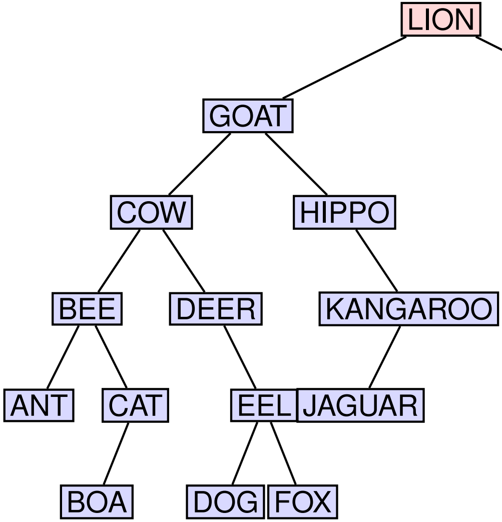
Let us consider now the search algorithm in a BST. Because of their recursive definition of BST, we will start with a recursive algorithm.
Let T be a BST representing the dictionary and let k be the key we are looking for. If $T = \Box$ then k is not in the dictionary and we need only to signal this event in some convenient way (e.g. we return `false`). If T is not empty we will need to check the relation between k and the key of the element x stored at the root of T.

Search

If $k = \text{KEY}(x)$ the search is successful and we can stop it there, returning x or the information associated to x which we care about. If $k < \text{KEY}(x)$, following the definition of BSTs, if there exists an element in T with key k it must be stored in the left subtree of T; hence, we must make a recursive call on the left subtree of T to continue the search. Analogously, if $k > \text{KEY}(x)$ then the search must recursively continue in the right subtree of T.

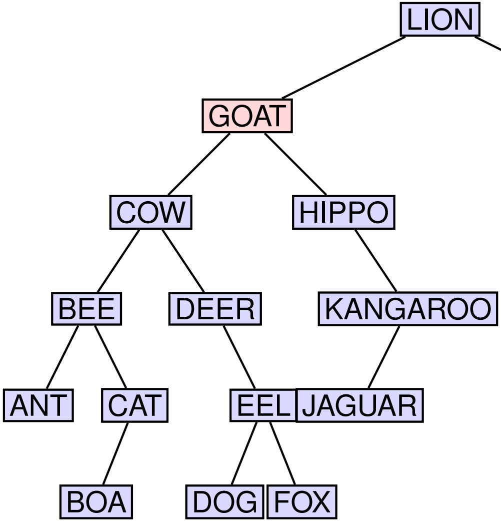
Search: An Example

Searching for DOG



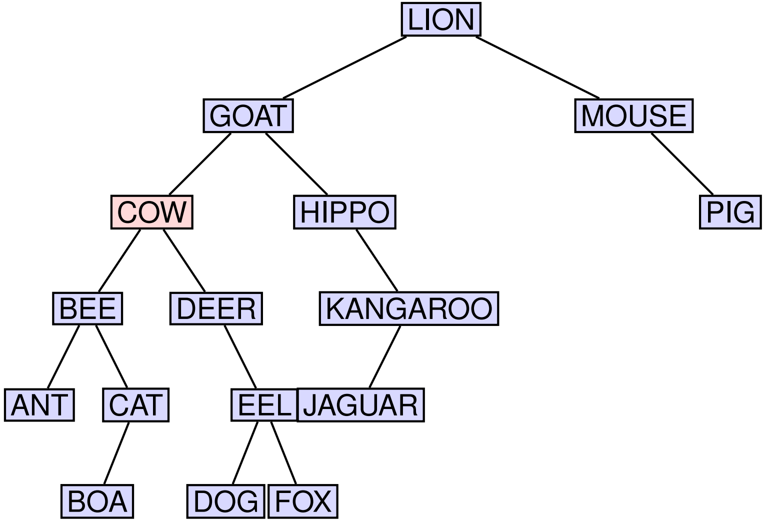
Search: An Example

Searching for DOG



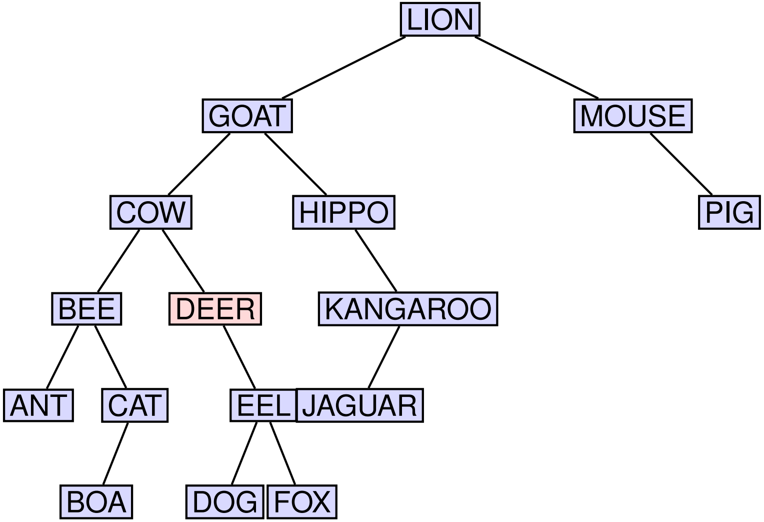
Search: An Example

Searching for DOG



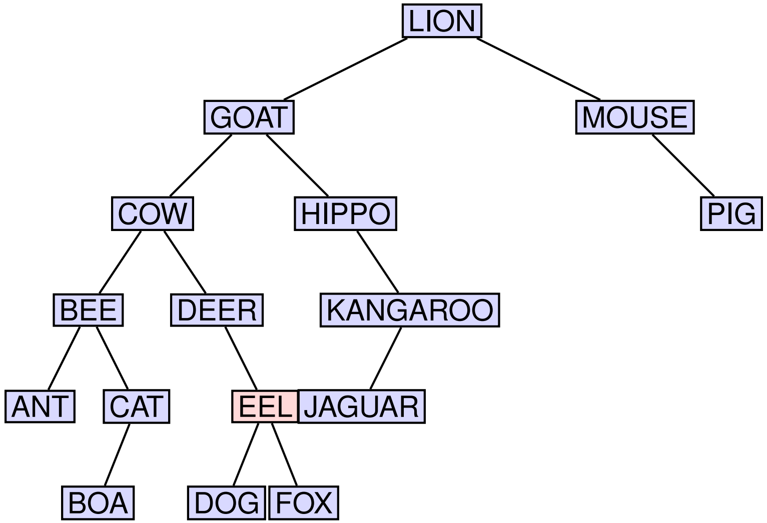
Search: An Example

Searching for DOG



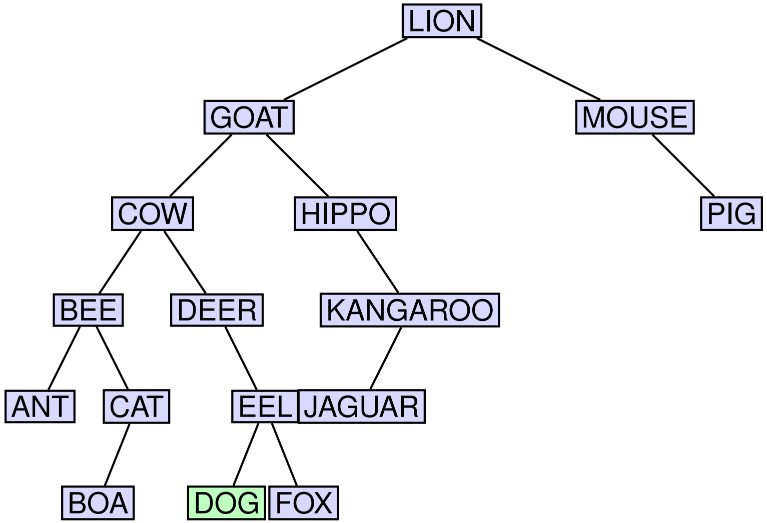
Search: An Example

Searching for DOG



Search: An Example

Searching for DOG



Search

The public method LOOKUP uses the private class method BST_LOOKUP. This one receives a pointer p to the root of the BST where we have to perform the search for the key k. It returns a pointer, either a null pointer if the search is unsuccessful or a pointer to the node that contains the element with the sought key.

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::lookup(const Key& k,
   bool& exists, Value& v) const {

   node_bst* p = bst_lookup(root, k);
   if (p == nullptr)
      exists = false;
   else {
      exists = true;
      v = p -> _v;
   }
}
```

Search

The recursive implementation of BST_LOOKUP is almost immediate from the definition of BST. If the tree is empty or its root (the node pointed to by p) contains the given k we are done and return p; otherwise, we compare k with the key stored at the root node and continue either on the left or on the right, as necessary.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
   Dictionary<Key,Value>::bst_lookup(node_bst* p,
     const Key& k) {

   if (p == nullptr or k == p -> _k)
     return p;

   // p != nullptr and k != p -> _k
   if (k < p -> _k)
      return bst_lookup(p -> _left, k);
   else // p -> _k < k
      return bst_lookup(p -> _right, k);
}
```

Search

The tail recursion in the previous recursive implementation of BST_LOOKUP can be easily removed to get an iterative implementation.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
   Dictionary<Key,Value>::bst_lookup(node_bst* p,
     const Key& k) {

   while (p != nullptr and k != p -> _k) {
      if (k < p -> _k)
         p = p -> _left;
      else // p -> _k < k
         p = p -> _right;
   }
   return p;
}
```

Insertions

The insertion algorithm is also very easy and we can design it by using the same reasoning as for the search algorithm: if the new key to be inserted is smaller than the key at the root insert in the left subtree; otherwise, insert into the right subtree.
The public method INSERT relies on the private class method BST_INSERT. It gets the $\langle key, value\rangle$ to insert and a pointer to the root of the BST (the pointer is null when the tree is empty); the method returns a pointer to the root of the resulting tree. If the given key already appears in the tree then no new element is inserted, but the value associated to the key is changed to the new given value.

Insertions

First, a trivial implementation for the constructor of the class `node_bst`:

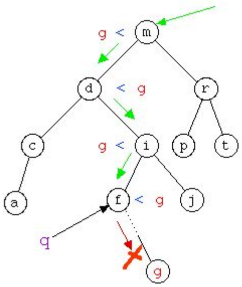
```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst(const Key& k,
   Value& v, node_bst* left, node_bst* right) :
   _k(k), _v(v), _left(left), _right(right) {
}
```

Insertions

```
template <typename Key, typename Value>
void Dictionary::insert(const Key& k,
     const Value& v) {

   root = bst_insert(root, k, v);

}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
   Dictionary<Key,Value>::bst_insert(node_bst* p,
      const Key& k, const Value& v) {

   if (p == nullptr)
      return new node_bst(k, v);

   // p != nullptr, continue the insertion in the
   // appropriate subtree or update the associated
   // value if p -> _k == k
   if (k < p -> _k)
      p -> _left = bst_insert(p -> _left, k, v);
   else if (p -> _k < k)
      p -> _right = bst_insert(p -> _right, k, v);
   else // p -> _k == k
      p -> _v = v;
   return p;
}
```

Insertions

The iterative implementation of the insertion is more complicated; besides locating the leaf (empty subtree) where the new elements must be inserted, we will need to keep a pointer f to the father of the current node. When the loop finds the empty subtree where the new element should be inserted, f points to the node that will become the father of the new node.



Insertions

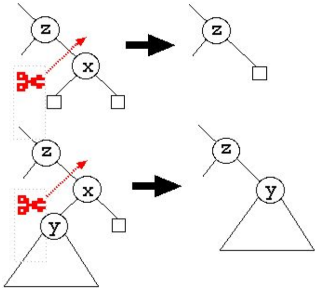
```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
   Dictionary<Key,Value>::bst_insert(node_bst* p,
      const Key& k, const Value& v) {

   // if the BST is empty
   if (p == nullptr)
      return new node_bst(k, v);

   // otherwise
   node_bst* f = nullptr;
   node_bst* root = p;

   // lookup the leaf where we have to insert (or
   // end in a node with key k to perform an update
   // of the associated value)
   while (p != nullptr and k != p -> _k) {
      f = p;
      if (k < p -> _k)
         p = p -> _left;
      else // p -> _k < k
         p = p -> _right;
   }

   // insert or update
   if (p == nullptr) { // insert a new node
      if (k < f -> _k)
         f -> _left = new node_bst(k, v);
      else // k > f -> _k
         f -> _right = new node_bst(k, v);
   }
   else // update value, k == p -> _k
      p -> _v = v;

   return root;
```

Deletions

Deleting an element with given key k involves two phases:

1. Locating the node with key k (if any) in the tree
2. Removing that node

The first phase is a lookup. For the second phase, if the node is a leaf (both subtrees are empty) then it suffices to remove the node. When the node x to be removed has only one non-empty subtree, it is not very difficult to remove the node: lift the non-empty subtree so that the father of x points to the subtree's root instead of pointing to x.



Deletions

The problematic situation is when the node x to be removed has two non-empty subtrees.
We will deal with all possible cases, even the complicated one, solving the following problem: given two BSTs T_1 and T_2 such that all keys in T_1 are smaller than all keys in T_2, join T_1 and T_2 into a single BST T with all the keys; we will write

$$T = \text{JOIN}(T_1, T_2).$$

Quite obviously:

$$\text{JOIN}(T, \square) = T,$$
$$\text{JOIN}(\square, T) = T.$$

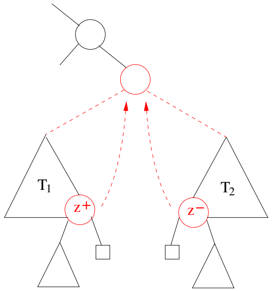
In particular, $\text{JOIN}(\square, \square) = \square$.

Deletions

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::remove(
     const Key& k) {

   root = bst_remove(root, k);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
   Dictionary<Key,Value>::bst_remove(node_bst* p,
      const Key& k) {

   // key k is not in the tree if it is empty
   if (p == nullptr) return p;

   if (k < p -> _k)
      p -> _left = bst_remove(p -> _left, k);
   else if (p -> k < k)
      p -> _right = bst_remove(p -> _right, k);
   else { // k == p -> k
      node_bst* to_kill = p;
      p = join(p -> _left, p -> _right);
      delete to_kill;
   }
   return p;
}
```

Deletions

Let z^+ be the largest key in T_1. Since it is larger than the other keys in T_1 but smaller than any key in T_2, we can join T_1 and T_2 putting z^+ as the root of T, T_2 as its right subtree, and the result of removing z^+ from T_1 —call it T_1'— as the left subtree.
The good news is that since z^+ has the largest key in T_1, the corresponding node cannot have a non-empty right subtree: therefore it is trivial to remove z^+ from T_1.



Deletions

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
  Dictionary<Key,Value>::join(node_bst* t1,
     node_bst* t2) {

   // trivial if one or two of the trees are empty
   if (t1 == nullptr) return t2;
   if (t2 == nullptr) return t1;

   // t1 != nullptr and t2 != nullptr
   node_bst* z = relocate_max(t1);
   z -> _right = t2;
   return z;

   // alternative: z = relocate_min(t2);
   //               z -> _left = t1;
   //               return z;
}
```

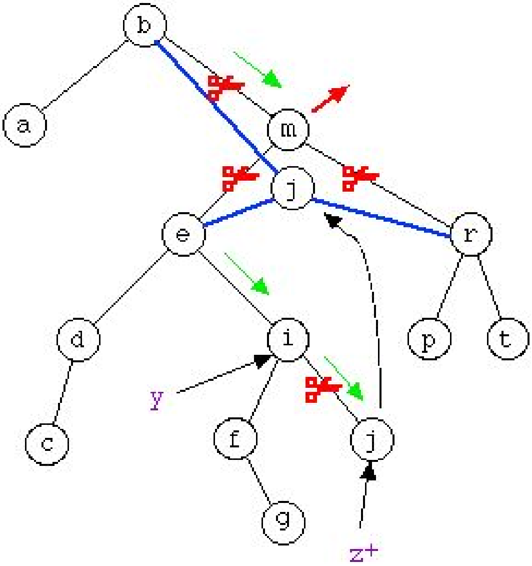
Deletions

The private class method RELOCATE_MAX receives a pointer p to the root of some BST T and it returns a pointer to the root of a new BST T'. That root contains the element with largest key in T, the right subtree of T' is empty and the left subtree of T' contains all the keys in T except the largest one which has been uplifted to the root.

There is only one tricky situation to deal with: when the largest key of T is already in the root of T; if such situation is detected the method has nothing more to do.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
   Dictionary<Key,Value>::relocate_max(
      node_bst* p) {

   node_bst* f = nullptr;
   node_bst* orig_p = p;
   while (p -> _right != nullptr) {
      f = p;
      p = p -> _right;
   }
   if (f != nullptr) {
      f -> _right = p -> _left;
      p -> _left = orig_p;
   }
   return p;
}
```

Deletions



Deletions

The joining of two trees can also be done by “relocating” the element with smallest key in T_2 to become the common root. Experimental work has shown that it is better to alternate between both versions of join (for instance, making a random choice or using one version or the other on an alternating basis). Using systematically either of the two variants leads to significantly biased and unbalanced search trees in the long run, even if the insertions are “random”.

The Cost of Searches and Updates in BSTs

A BST of size n can be equivalent to a linked lists, with one node with no children and $n-1$ nodes with only one non-empty subtree. Such a tree can be built by inserting n elements in ascending or descending order of their keys into an initially empty BST.

Thus in the height of a BST can be as high as n; in the opposite extreme a perfectly balanced BST of size n has height at least $\lceil \log_2(n+1) \rceil$.

For a BST of height h the worst-case cost of a search, an insertion or a deletion has cost $\Theta(h)$. Hence, the worst-case for seach and update operations in a BST of size n is $\Theta(n)$.

The Cost of Searches and Updates in BSTs

However, on average, the height of a random BST is $\Theta(\log n)$, and thus the average cost of the basic operations is $\Theta(\log n)$ too.

By a random BST of size n, we mean a tree built by n insertions, with all possible $n!$ orderings of the n insertions being equally likely.

For successful searches we will assume that the sought element is any of the n elements in the tree; for unsuccessful searches (and insertions) we will assume that the search ends at any of the $n + 1$ empty subtrees with identical probability.

The Cost of Searches and Updates in BSTs

Moreover, we will consider only the number of comparisons between the given key and the keys of the nodes that we examine. The cost of the operation will be proportional to this number of comparisons. Let $C(n)$ be the expected number of comparisons made by a succesful search on a random BST of size n, and $C(n, k)$ the expected number of comparisons conditioned to the root storing the element with the k-th smallest key. Then

$$C(n) = \sum_{1 \leq k \leq n} C(n; k) \times \mathbb{P}[\text{root is the } k\text{-th}] \,.$$

The Cost of Searches and Updates in BSTs

$$
\begin{aligned}
C(n) &= \frac{1}{n} \sum_{1\le k\le n} C(n;k) \\
&= 1 + \frac{1}{n} \sum_{1\le k\le n} \left(\frac{1}{n}\cdot 0 + \frac{k-1}{n}\cdot C(k-1) + \frac{n-k}{n}\cdot C(n-k) \right) \\
&= 1 + \frac{1}{n^2} \sum_{0\le k<n} \left(k\cdot C(k) + (n-1-k)\cdot C(n-1-k)\right) \\
&= 1 + \frac{2}{n^2} \sum_{0\le k<n} k\cdot C(k).
\end{aligned}
$$

An alternative way to analyze the cost of successful searches is to consider the so-called Internal Path Length (IPL). Given a BST, its IPL is the sum of the depths from the root to every other node in the tree. The *External Path Length* (EPL) is defined similarly and it is used in the analysis of unsuccessful searches/insertions.

The Cost of Searches and Updates in BSTs

If $I(n)$ denotes the expected value of the IPL of a random BST of size n then

$$C(n) = 1 + \frac{I(n)}{n}$$

The expected value of the IPL satisfies the following recurrence:

$$I(n) = n - 1 + \frac{2}{n} \sum_{0 \leq k < n} I(k), \qquad I(0) = 0. \tag{1}$$

Indeed, every node except the root of the tree contributes to the total IPL one unit plus its contribution to the IPL of the subtree of the root to which the node belongs.

Note also that the IPL of a tree does not only give us the cost of succesful searches in that tree; it is also the cost of building that tree starting from an empty tree.

The Cost of Searches and Updates in BSTs

Recurrence (1) is identical to the recurrence for the cost of quicksort!!
That's no coincidence. The recursion tree associated to a quicksort execution is a BST of size n: each node holds the pivot used at the corresponding recursive call. Now, every element in the tree has been compared (not being the pivot) against every pivot selected in the sequence of recursive calls leading from the initial call to the recursive call where the element is finally choosen as the picot. That is, every element is compared to all its proper ancestors in the BSTs, or in other terms, the number of comparisons it gets involved (not as a pivot) is its depth in the BST. Hence $I(n) = Q(n)$, where $Q(n)$ is the expected number of comparisons made by quicksort to sort an array of size n.

The Cost of Searches and Updates in BSTs

In what follows we provide a solution of the recurrence (although we had already solved it using CMT). Our first step is to compute $(n+1)I(n+1) - nI(n)$:

$$\begin{aligned}(n+1)I(n+1) - nI(n) &= (n+1)n - n(n-1) + 2I(n) \\ &= 2n + 2I(n); \\ (n+1)I(n+1) &= 2n + (n+2)I(n)\end{aligned}$$

The Cost of Searches and Updates in BSTs

$$\begin{aligned}
I(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1} I(n) = \frac{2n}{n+1} + \frac{2(n-1)(n+2)}{n(n+1)} + \frac{n+2}{n} I(n-1) \\
&= \frac{2n}{n+1} + \frac{2(n-1)(n+2)}{n(n+1)} + \frac{2(n-2)(n+2)}{n(n-1)} + \frac{n+2}{n-1} I(n-2) \\
&= \frac{2n}{n+1} + 2(n+2) \sum_{i=1}^{i=k} \frac{n-i}{(n-i+1)(n-i+2)} + \frac{n+2}{n-k+1} I(n-k) \\
&= \frac{2n}{n+1} + 2(n+2) \sum_{i=1}^{i=n} \frac{i}{(i+1)(i+2)} \\
&= \mathcal{O}(1) + 2(n+2) \sum_{1 \le i \le n} \left(\frac{2}{i+2} - \frac{1}{i+1} \right) \\
&= \mathcal{O}(1) + 2(n+2)(\frac{2}{n+2} + \frac{1}{n+1} + H_n - 2) \\
&= 2nH_n - 4n + 4H_n + \mathcal{O}(1),
\end{aligned}$$

The Cost of Searches and Updates in BSTs

where $H_n = \sum_{1 \le i \le n} 1/i$.

Since $H_n = \ln n + \mathcal{O}(1)$ we have

$$\begin{aligned} I(n) = Q(n) &= 2n \ln n + \mathcal{O}(n) \\ &= 1.386\ldots n \log_2 n + \mathcal{O}(n) \end{aligned}$$

For a successful search in a BST of size n, the average number of comparisons is then

$$C(n) = 1 + \frac{I(n)}{n} = 2 \ln n + \mathcal{O}(1).$$

Other Operations

Most other operations in BSTs are also very simple to implement and have good expected cost. To achieve logarithmic expected cost in searches and deletions by rank, or in counting operations it is necessary to modify the standard implementation of BSTs (their representation and the basic operations) so that each node in the BST holds updated information about the size of the subtree rooted at that node.

We conclude with a particular example here: given two keys k_1 and k_2 with $k_1 < k_2$ we implement a new method `in_range` which returns a list of the elements in the BST (actually a list of pairs $\langle key, value\rangle$) which key k is within the range $[k_1 \ldots k_2]$, that is, $k_1 \leq k \leq k_2$.

Other Operations: `in_range`

If the BST is empty, the result is an empty list. If the BST is not empty, let k be the key at the root. If $k < k_1$ then no element in the left subtree will be returned and no recursive call on the left subtree should be made. Likewise, if $k_2 < k$, no element in the right subtree is within the range and we have thus to avoid the recursive call in the right subtree.
If $k_1 \leq k \leq k_2$ then the element at the root is within the range and must be added to the result; moreover, elements from both the left and the right subtrees might be within the range too and thus both subtrees must be recursively explored. To make sure that the returned list is in ascending order of keys, we must first make the recursive call in the left subtree, then add the root eleemnt to the list, finally make the recursive call into the right subtree.

It can be shown that the average cost is $\Theta(\log n + F)$, where F is the length of the result (it can go from $\Theta(1)$ when k_1 and k_2 aren't too far apart, to $\Theta(n)$ if every key is within the range).

Other Operations: `in_range`

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::in_range(
     const Key& k1, const Key& k2,
     list<pair<Key,Value> >& result) const {

        bst_in_range(root, k1, k2, result);
}

template <typename Key, typename Value>
void Dictionary<Key,Value>::bst_in_range(node_bst* p,
    const Key& k1, const Key& k2,
    list<pair<Key,Value> >& result) const {

    if (p == nullptr) return;
    if (k1 <= p -> _k)
      bst_in_range(p -> _left, k1, k2, result);

    if (k1 <= p -> _k and p -> _k <= k2)
      result.push_back(make_pair(p -> _k, p -> _v));

    if (p -> _k <= k2)
      bst_in_range(p -> _right, k1, k2, result);
}
```

Part III

Dictionaries

Balanced Trees

The main drawback of standard BSTs is that they can be strongly unbalanced.

In the worst-case, the height of BST of size n is $\Theta(n)$, hence the worst-case cost of search, insertion and deletion is $\Theta(n)$ too.

Several alternatives have been proposed to overcome this problem: the oldest and possibly one of the simplest and most elegant solution is height balancing, originally proposed by the Russian computer scientists Adelson-Velskii and Landis. These search trees are named AVLs after their inventors.

AVLs

Definition

An AVL T is a binary search tree that is either empty or

1. Its left and right subtrees, L and R, respectively are AVLs
2. The height of L and R differs by one at most:

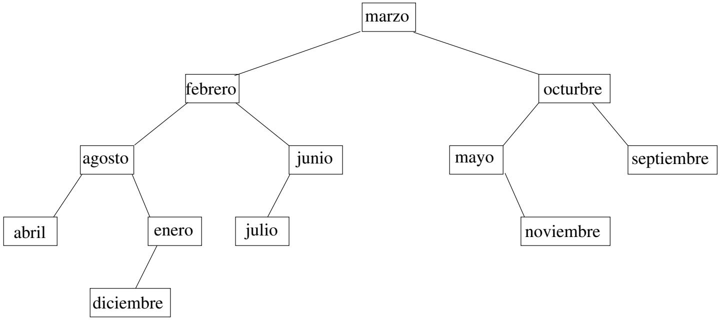
$$|\text{height}(L) - \text{height}(R)| \leq 1$$

AVLs

The previous recursive definition guarantees that the balance invariant holds for all nodes (roots of the corresponding subtrees) of an AVL. If we denote bal(x) the difference between the height of the left and right subtrees of an arbitrary node x, then we must have bal$(x) \in \{-1, 0, +1\}$.
Since AVLs are BSTs, the search algorithm in an AVL is exactly the same as the algorithm for standard BSTs; moreover, an inorder traversal of an AVL cisits its nodes in ascending order of the keys.

AVLs

{enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre noviembre, diciembre}



The Cost of AVLs

Lemma

The height $h(T)$ of an AVL T of size n is $\Theta(\log n)$.

Proof. Since any AVL is a binary tree we have $h(T) \geq \lceil \log_2(n+1) \rceil$, that is, $h(T) \in \Omega(\log n)$. We need just to show that $h(T) \in \mathcal{O}(\log n)$.
Let N_h the least number of nodes we need to build an AVL of size h. Clearly, $N_0 = 0$ and $N_1 = 1$. The most unbalanced possible AVL with height $h > 1$ can be build by putting a root then a left subtree which is a most unbalanced AVL of height $h-1$ (using only N_{h-1} nodes), and a right subtree which is a most unblanced AVL of height $h-2$ (using N_{h-2}). Hence

$$N_h = 1 + N_{h-1} + N_{h-2}$$

The Cost of AVLs

A few values of the sequence $\{N_h\}_{h\geq 0}$:

$$0, 1, 2, 4, 7, 12, 20, 33, 54, 88, \ldots$$

Its similiraty to the Fibonacci sequence draws our attention. It is not difficult to prove (for instance, by induction) that $N_h = F_{h+1} - 1$ for all $h \geq 0$. Indeed, $N_0 = F_1 - 1 = 1 - 1 = 0$ and

$$N_h = 1 + N_{h-1} + N_{h-2} = 1 + (F_h - 1) + (F_{h-1} - 1) = F_{h+1} - 1$$

The n-th Fibonacci number is

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor,$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.61803\ldots$ denotes the *golden ratio*.

The Cost of AVLs

Consider now an AVL of size n and height h. Then we must have $n \geq N_h$ by definition, and

$$n \geq F_{h+1} - 1 \geq \frac{\phi^{h+1}}{\sqrt{5}} - \frac{3}{2}$$

Hence

$$(n + \frac{3}{2})\frac{\sqrt{5}}{\phi} \geq \phi^h$$

Taking logarithms base ϕ, and after simplification

$$h \leq \log_\phi n + \mathcal{O}(1) = 1.44 \log_2 n + \mathcal{O}(1)$$

Updating AVLs

The previous lemma guarantees that the cost of any search in an AVL of size n is $\mathcal{O}(\log n)$, even in the worst case.
The problem we face now is how to maintain the balance invariant of AVLs after insertions and deletions.
Both insertions and deletions act as their counterparts for standard BSTs, but all the nodes in the path from the root to the ins/del point must be checked for balance; if the operation invalidates the balance condition at some node x, we must do something to fix it and reestablish the invariant.
The first observation is that we will need to store at each node information about its height or its balance (bal$(\cdot)$) to avoid costly computations.

Implementation of AVLs

```
template <typename Key, typename Value>
class Dictionary {
public:
  ...
  void lookup(const Key& k,
       bool& exists, Value& v) const;
  void insert(const Key& k,
       const Value& v);
  void remove(const Key& k);
  ...
private:
  struct node_avl {
     Key _k;
     Value _v;
     int _height;
     node_avl* _left;
     node_avl* _right;
     // constructor for class node_avl
     node_avl(const Key& k, const Value& v,
              int height = 1,
              node_avl* left = nullptr,
              node_avl* right = nullptr);
  };
  node_avl* root;
  ...
  static int height(node_avl* p);
  static void update_height(node_avl* p);
};
```

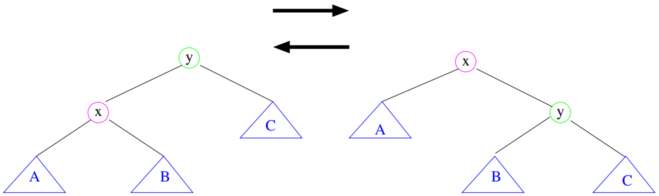
Implementation of AVLs

```
int max(int x, int y) {

   return x > y ?  x : y;
}

template <typename Key, typename Value>
static int Dictionary<Key,Value>::height(
              node_avl* p) {

   if (p == nullptr)
      return 0;
   else
      return p -> _height;
}

template <typename Key, typename Value>
static void Dictionary<Key,Value>::update_height(
              node_avl* p) {

   p -> _height = 1 + max(height(p -> _left), height(p -> _right));
}
```

Rotations

To reestablish the balance invariant in a node where it didn't hold we will use rotations.

Figure below shows the simplest rotations. Note that if the tree on the left is a BST ($A < x < B < y < C$) then the tree on the right is a BST too.



Rotations

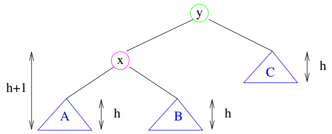
Assume there is an AVL subtree with root y and that we are inserting a key k such that $k < x < y$; once the insertion is made, assume that y gets unbalanced.
If the height of C is h then the subtree rooted at x must have height h or height $h + 1$. If the height of x was h then the insertion of the new key k cannot make y unbalanced, hence we must assume that the height of x is $h + 1$.

Rotations

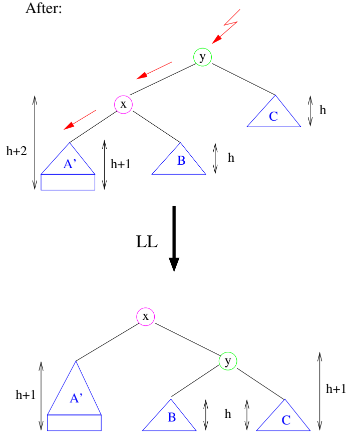
Analogous reasonings lead us to conclude that the height of A must be h, hence the height of y before the insertion is $h + 2$. Assume now that the height of A increases as a consequence of the insertion, that is, the resulting subtree A' has height $h + 1$. If we assume that x is still balanced after the insertion (but y does not) we must conclude that B has height h too. After the insertion the subtree rooted at x increases its height to $h + 2$ and thus bal$(y) = +2$!

Rotations

Before:



After:



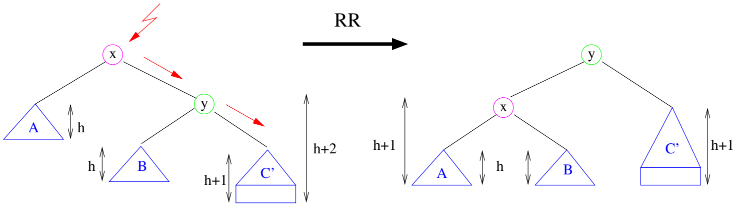
Rotations

Applying the rotation to node y we preserve the BST property but we also reestablish the AVL condition on y. By hypothesis, A', B are C are AVLs. After the rotation the height of y is $h+1$ and its balance is 0, and the height of x changes to $h+2$ and its balance is 0 too.

We have not just solved the unbalance of y: the subtree where we continued with the insertion is an AVL and its height is the same as the height of the subtree prior to the insertion, which means that no ancestor of y in the AVL will be unbalanced, once the unbalance has been fixed.
The rotation that we have used in this example is often called simple rotation LL (left-left).

Rotations

An analogous analysis reveals that when the new inserted key k satisfies $x < y < k$ we maybe unbalance the node x; this is unbalance is corrected applying a simple rotation RR (right-right) on node x. Rotation RR is the symmetric of a rotation LL.

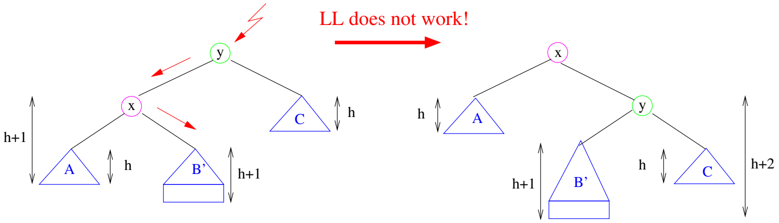


Rotations

We analyze now the case where we have an AVL subtree with root y and we are making the insertion of a key k with $x < k < y$. Suppose that once k has been inserted, node y gets unbalanced.
Reasoning as before we conclude that if height$(C) = h$ then the height of the subtree rooted at x must be $h + 1$ and height$(B) = h$.
Let B' be the result of inserting the new key into B. Suppose that the height of B increases after the insertion, i.e., height$(B') = h + 1$. If x is still balanced, but y does not, we conclude that height$(x) = h$. After the insertion, the height of the subtree rooted at x changes to $h + 2$ and hence bal$(y) = +2$!

Rotations

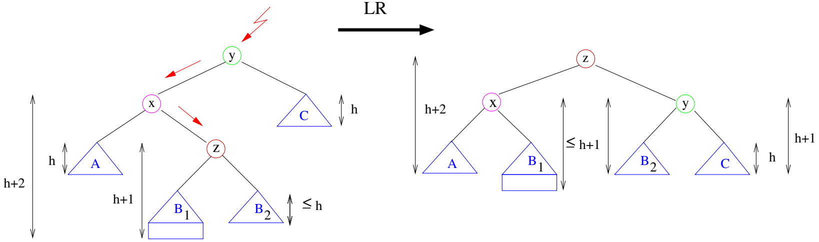
If we apply a simple rotation LL on node y, the BST condition is preserved but the AVL condition is not satisfied. Indeed, by hypothesis, A, B' and C areAVLs. After a rotation LL height$(y) = h + 2$ and bal$(y) = +1$; but the height$(x) = h + 3$ and bal$(x) = -2$!



We cannot solve the problem with simple rotations. We must figure out something else.

Rotations

Suppose the root of B' (B' = the result of insertion into B) is z and that the subtrees of z are the two AVLs B_1 and B_2. Since height$(B') = h + 1$ yand B' is an AVL, at least one subtree B_i has height $= h$ and the other B_j has height $= h$ or $= h - 1$. We will apply a rotation bringing z to the root, with x the root of the left subtree and subtrees A and B_1, and y the root of the right subtree with children B_2 and C.



Note that inorder traversal before and after the rotation remains the same; the new type of rotation preserves the BST property

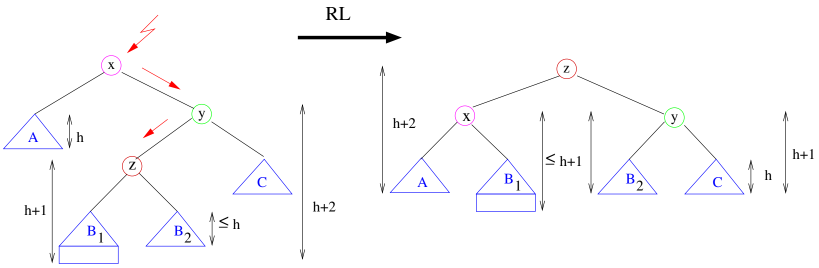
Rotations

The rotation on node y preserves the BST property, but it also reestablishes the balance at node y. By hypothesis, A, B and C are AVLs. After rotation the height of the subtree rooted at x is $h + 1$ and its balance is 0 or $+1$, the height of y is $h + 1$ and its balance 0 or -1, and height(z) becomes $h + 2$ and the balance continues to be 0.

Similar to rotations LL and RR, the application of the double rotation LR (left-right) on node y reestablishes its balance, we now that no further violations of the AVL condition will hapen in any ancestor od y.

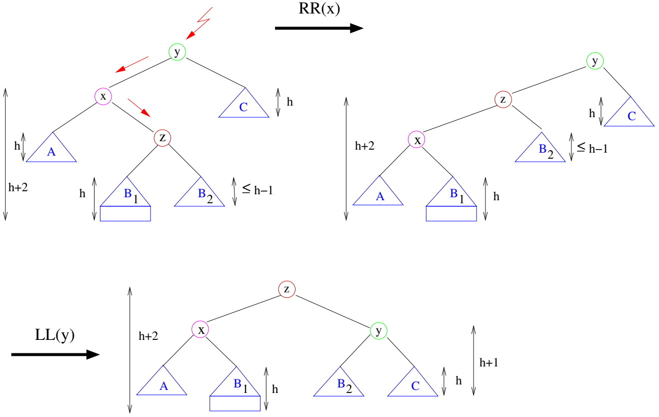
Rotations

When a new keys goes first to the right, then to the left of a node, the unbalance at the node (if it happens) is corrected with a double rotation RL, which is symmetric to a rotation LR.



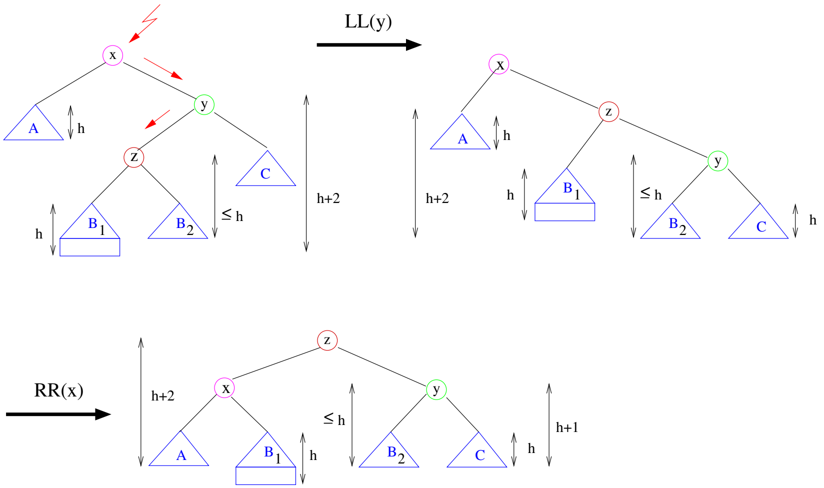
Rotations

Double rotations LR and RL are called double because we can decompose each as a sequence of two simple rotations. For instace, a double rotation LR on node y is equivalent to applying a rotation RR on x followed by a rotation LL on y (note that its left subtree is, after the simple rotation, z, not x).



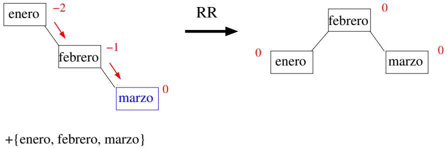
Rotations

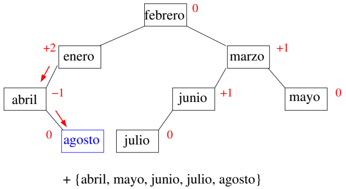
A rotation RL is the sequential composition of a rotation LL, followed by a rotation RR.

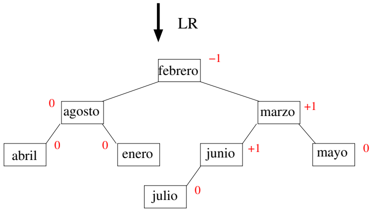


Insertions in AVLs

Example of an AVL storing the Spanish names of the months:

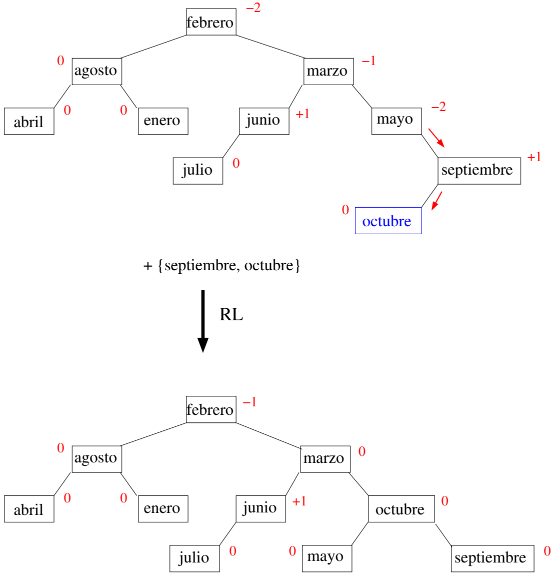






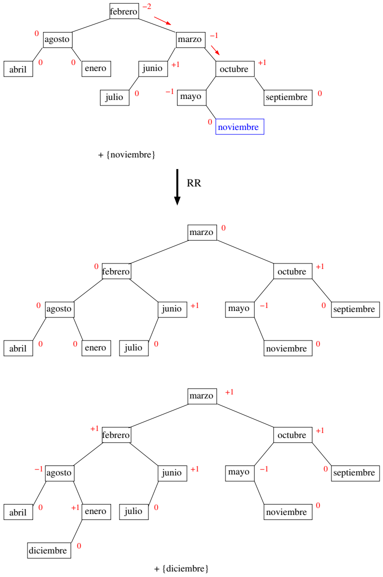
Insertions in AVLs

Example of an AVL storing the Spanish names of the months:



Insertions in AVLs

Example of an AVL storing the Spanish names of the months:



Insertions in AVLs

Facts:

1. A rotation (simple or double) does only involve update a few pointers and the update of the `_height` fields, and its cost is $\Theta(1)$, independent of the tree's size.
2. During an insertion in an AVL we will need to perform a rotation at most to reestablish the AVL condition. The rotation is applied (if needed) to a node in the path from the root to the insertion point.
3. The worst-case cost of an insrtion in an AVL is $\Theta(\log n)$.

Insertions in AVLs

```
template <typename Key, typename Value>
class Dictionary {
public:
  ...
private:
  struct node_avl {
       ...
  };
  node_avl* root;
  ...
  static node_avl* avl_insert(node_avl* p,
    const Key& k, const Value& v);
  static node_avl* rotate_LL(node_avl* p);
  static node_avl* rotate_LR(node_avl* p);
  static node_avl* rotate_RL(node_avl* p);
  static node_avl* rotate_RR(node_avl* p);
  ...
};
```

Insertions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::rotate_LL(node_avl* p) {
     node_avl* q = p -> _left;
     p -> _left = q -> _right;
     q -> _right = p;
     update_height(p);
     update_height(q);
     return q;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::rotate_RR(node_avl* p) {
     node_avl* q = p -> _right;
     p -> _right = q -> _left;
     q -> _left = p;
     update_height(p);
     update_height(q);
     return q;
}
```

Insertions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::rotate_LR(node_avl* p) {
     p -> _left = rotate_RR(p -> _left);
     return rotate_LL(p);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::rotate_RL(node_avl* p) {
     p -> _right = rotate_LL(p -> _right);
     return rotate_RR(p);
}
```

Insertions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::avl_insert(node_avl* p,
      const Key& k, const Value& v) {

   if (p == nullptr)
      return new node_avl(k, v);

   if (k < p -> _k)  {
      p -> _left = avl_insert(p -> _left, k, v);
      // chec balance at p and rotate if needed
      if (height(p -> _left) - height(p -> _right) == 2) {
         // p -> _left cannot be empty
         if (k < p -> _left -> _k) // LL
            p = rotate_LL(p);
         else // LR
            p = rotate_LR(p);
      }
   }
   else if (p -> _k < k) { // symmetric case
      p -> _right = avl_insert(p -> _right, k, v);
      if (height(p -> _right) - height(p -> _left) == 2) {
         if (p -> _right -> _k < k) // RR
            p = rotate_RR(p);
         else // RL
            p = rotate_RL(p);
      }
   }
   else // p -> _k == k
      p -> _v = v;

   update_height(p);

   return p;
}
```

Insertions in AVLs

We can write an iterative version of the AVL insertion algorithm, but it is quite complicated. Once the new item is inserted to replace a leaf (empty subtree) we must unroll the followed path to check if there is some node there which got unbalanced and apply the corresponding rotation.
A recursive implementation does the unrolling of the search path “for free”, after returnining from a recursive call we need just to check if the current node is still balanced or not and act accordingly. For an iterative implementation we would need either: 1) explicit pointers to the parent; or 2) store the path from the root in a stack, and pop elements (=pointers to nodes) from the stack to unroll the path; or 3) more exotic alternatives such as *pointer reversal*.

Deletions in AVLs

The deletion algorithm for AVLs draws upon the same ideas, although the analysis to determine what rotation must be applied is a bit more complex.
The worst-case of deletions in AVLs is $\Theta(\log n)$. Unbalance in a node is corrected by a simple or a double rotation, but it might be necessary to apply several rotations to fix the balance. However, not more than a rotation is necessary in any given node in the path from the root to the deletion point, since there are only $\mathcal{O}(\log n)$, the worst-case is $\Theta(\log n)$.
The algorithm is slightly more complicated than that of insertions; the iterative version is still more complicated.

Deletions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::avl_remove(node_avl* p,
      const Key& k) {
   if (p == nullptr) return p;
   if (k < p -> _k)  {
      p -> _left = avl_remove(p -> _left, k);
      // check balance and rotate if needed
      if (height(p -> _right) - height(p -> _left) == 2) {
         // p -> _right cannot be empty
         if (height(p -> _right -> _left)
              - height(p -> _right -> _right) == 1)
            p = rotate_RL(p);
         else
            p = rotate_RR(p);
      }
   }
   else if (p -> _k < k) { // symmetric case
      p -> _right = avl_remove(p -> _right, k);
      if (height(p -> _left) - height(p -> _right) == 2) {
         if (height(p -> _left -> _right)
              - height(p -> _right -> _left) == 1)
            p = rotate_LR(p);
         else
            p = rotate_LL(p);
      }
   }
   else { // p -> _k == k
      node_avl* to_kill = p;
      p = join(p -> _left, p -> _right);
      delete to_kill;
   }
   update_height(p);
   return p;
}
```

Deletions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
  Dictionary<Key,Value>::join(node_avl* t1,
     node_avl* t2) {

   // trivial, if one of the trees is empty
   if (t1 == nullptr) return t2;
   if (t2 == nullptr) return t1;

   // t1 != nullptr and t2 != nullptr
   node_alv* z;
   remove_min(t2, z);
   z -> _left = t1;
   z -> _right = t2;
   update_height(z);
   return z;
}
```

Deletions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
   Dictionary<Key,Value>::remove_min(
      node_avl*& p, nod_avl*& z) {

   if (p -> _left != nullptr) {
      remove_min(p -> _left, z);
      // check balance and rotate if needed
      if (height(p -> _right) - height(p -> _left) == 2) {
         // p -> _right cannot be empty
         if (height(p -> _right -> _left)
              - height(p -> _right -> _right) == 1)
            p = rotate_RL(p);
         else
            p = rotate_RR(p);
      }
   } else {
     z = p;
     p = p -> _right;
  }
  update_height(p);
}
```

Part III

Dictionaries

Hash Tables

A hash table (cat: *taula de dispersió*, esp: *tabla de dispersión*) allows us to store a set of elements (or pairs $\langle key, value\rangle$) using a hash function $h : K \Longrightarrow I$, where I is the set of indices or addresses into the table, e.g., $I = [0..M - 1]$.

Ideally, the hash function h would map every element (their keys) to a distinct address of the table, but this is hardly possible in a general situation, and we should expect to find collisions (different keys mapping to the same address) as soon as the number of elements stored in the table is $n = \Omega(\sqrt{M})$.

Hash Tables

If the hash function evenly “spreads” the keys, the hash table will be useful as there will be a small number of keys mapping to any given address of the table.

Given two distinct keys x and y, we say that they are synonyms, also that they collide if $h(x) = h(y)$.

A fundamental problem in the implementation of a dictionary using a hash table is to design a collision resolution strategy.

Hash Tables

```
template <typename T> class Hash {
public:
  int operator()(const T& x) const ;
};

template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
public:
   ...
private:
   struct node {
       Key _k;
       Value _v;
       ...
   };
   nat _M; // capacity of the table
   nat _n; // number of elements (size) of the table
   double _alpha_max; // max. load factor
   HashFunct<Key> h;

   // open addressing
   node* _Thash; // an array with pairs <key,value>

   // separate chaining
   // node** _Thash; // an array of pointers to linked lists of synonyms

   int hash(const Key& k)  {
        return h(k) % _M;
   }
};
```

Hash Functions

A good hash function h must enjoy the following properties

1. It is easy to compute
2. It must evenly spread the set of keys K: for all i, $0 \leq i < M$

$$\frac{\#\{k \in K \mid h(k) = i\}}{\#\{k \in K\}} \approx \frac{1}{M}$$

Hash Functions

Defining a good hash function is not an easy task and it requires a solid background in several branches of Math. This task is very closely related to the definition of good (pseudo)random number generators.
As a general rule, the keys are first converted to positive integers (reading the binary representation of the key as a number), some mathematical transformation is applied, taking the result modulo M (the size of the table). For various theoretical reason, it is a good idea that M is a prime number.

Hash Functions

In our implementation, the class `Hash<T>` overloads operator `()` so that for an object `h` of the class `Hash<T>`, `h(x)` is the result of “applying” h to the object `x` of class `T`. The operation returns a positive integer.
The private method `hash` in class Dictionary computes

```
h(x) % _M
```

to obtain a valid position into the table, an index between 0 and `_M - 1`.

Hash Functions

```
// specialization of the template for T = string
template <> class Hash<string> {
public:
  int operator()(const string& x) const  {

     int s = 0;
     for (int i = 0; i < x.length(); ++i)
         s = s * 37 + x[i];
     return s;
};

// specialization of the template for T = int
template <> class Hash<int> {
static long const MULT = 31415926;
public:
  int operator()(const int& x) const  {

     long y = ((x * x * MULT) << 20) >> 4;
     return y;
  }
};
```

Other sophisticated hash functions use weighted sums or non-linear transformations (e.g., they square the number represented by the k central bits of the key).

Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- Open hashing: separate chaining, coalesced hashing, ...
- Open addressing: linear probing, double hashing, quadratic hashing, ldots

Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- Open hashing: separate chaining, coalesced hashing, …
- Open addressing: linear probing, double hashing, quadratic hashing, ldots

Separate Chaining

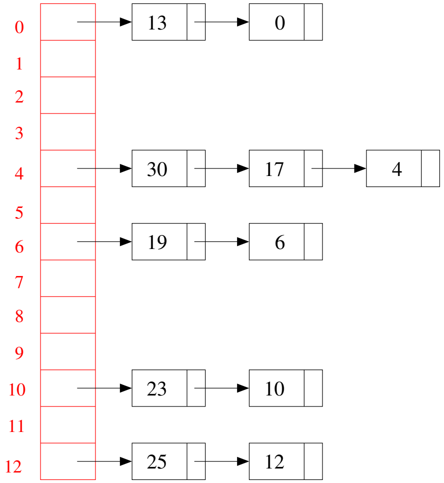
In separate chaining, each slot in the hash table has a pointer to a linked list of synonyms.

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
   ...
private:
   struct node {
      Key _k;
      Value _v;
      node* _next;
      // constructor for class node
      node(const Key& k, const Value& v,
           node* next = nullptr);
   };
   node** _Thash; // array of pointers to linked lists of synonyms
   int _M;        // capacity of the table
   int _n;        // number of elements
   double _alpha_max;  // max. load factor

   node* lookup_sep_chain(const Key& k, int i) const ;
   void insert_sep_chain(const Key& k,
           const Value& v);
   void remove_sep_chain(const Key& k) ;
};
```

Separate Chaining

M = 13 X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30}

h (x) = x mod M



Separate Chaining

For insertions, we access the apropriate linked list using the hash function, and scan the list to find out whether the key was already present or not. If present, we modify the associated value; if not, a new node with the pair $\langle key, value\rangle$ is added to the list.
Since the lists contain very few elements each, the simplest and more efficient solution is to add elements to the front.
There is no need for double links, sentinels, etc. Sorting the lists or using some other sophisticated data structure instead of linked lists does not report real practical benefits.

Separate Chaining

Searching is also simple: access the apropriate linked list using the hash function and sequentially scan it to locate the key or to report unsuccessful search.

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert(const Key& k,
     const Value& v) {
   insert_sep_chain(k, v);
   if (_n / _M > _alpha_max)
      // the current load factor is too large, raise here an exception or
      // resize the table and rehash
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert_sep_chain(
           const Key& k, const Value& v) {
   int i = hash(k);
   node* p = lookup_sep_chain(k,i);
   // insert as first item in the list
   // if not present
   if (p == nullptr) {

      _Thash[i] = new node(k, v, _Thash[i]);
      ++_n;
   }
   else
      p -> _v = v;
}
```

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::lookup(const Key& k,
   bool& exists, Value& v) const {

   node* p = lookup_sep_chain(k, hash(k));
   if (p == nullptr)
     exists = false;
   else {
     exists = true;
     v = p -> _v;
  }
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
Dictionary<Key,Value,HashFunct>::node*
   Dictionary<Key,Value,HashFunct>::lookup_sep_chain(const Key& k,
                                                     int i) const  {

   node* p = _Thash[i];
   // sequential search in the i-th list of synonyms
   while (p != nullptr and p -> _k != k)
      p = p -> _next;

   return p;
}
```

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value, HashFunct>::remove(const
        Key& k)  {

   remove_sep_chain(k);
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::remove_sep_chain
     (const Key& k)  {

   int i = hash(k);
   node* p = _Thash[i];
   node* prev = nullptr; // predecessor of p

   // sequential search in the i-th list of synonyms
   while (p != nullptr and p -> _k != k) {
      prev = p;
      p = p -> _next;
   }

   // if p != nullptr, remove from the
   // i-th list; check whether p is the first
   // element in the list or not
   if (p != nullptr) {
      --_n;
      if (prev != nullptr)
         prev -> _next = p -> _next;
      else
         _Thash[i] = p -> _next;
      delete p;
   }
}
```

The Cost of Separate Chaining

Let n be the number of elements stored in the hash table. On average, each linked list contains $\alpha = n/M$ elements and the cost of lookups (either successful or unsuccessful), of insertions and of deletions will be proportional to α. If α is a small constant value then the cost of all basic operations is, on average, $\Theta(1)$. However, it can be shown that the expected length of the largest synonym list is $\Theta(\log n)$.
The value α is called load factor, and the performance of the hash table will be dependent on it.

Open Hashing

Other variants of open hashing (e.g., coalesced hashing) store the synonyms in a particular area of the hash table, often called the cellar. The positions corresponding to the cellar are not addressable, that is, the hash function never maps a key into that area.

Open Addressing

In open addressing, synonyms are stored in the hash table. Searches and insertions probe a sequence of positions until the given key or an empty slot is found. The sequence of probes starts in position $i_0 = h(k)$ and continues with $i_1, i_2, \ldots$ The different open addressing strategies use different rules to define the sequence of probes. The simplest one is linear probing:

$$i_1 = i_0 + 1, i_2 = i_1 + 1, \ldots,$$

taking modulo M in all cases.

Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>

class Dictionary {
   ...
private:
struct node {
   Key _k;
   Value _v;
   bool _free;
   // constructor for class node
   node(const Key& k, const Value& v, bool libre = true);
};
node* _Thash;  // array of nodes
int _M;        // capacity of the table
int _n;        // number of elements
double _alpha_max;  // max. load factor (must be < 1)


int lookup_linear_probing(const Key& k) const ;
void insert_linear_probing(const Key& k,
        const Value& v);
void remove_linear_probing(const Key& k) ;
};
```

Linear Probing

M = 13 X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30}

h (x) = x mod M (incremento 1)

0	0
1	
2	
3	
4	4
5	
6	6
7	
8	
9	
10	10
11	
12	12

+ {0, 4, 6, 10, 12}

0	0	occupied
1	13	occupied
2		free
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8		free
9		free
10	10	occupied
11	23	occupied
12	12	occupied

+ {13, 17, 19, 23}

0	0	occupied
1	13	occupied
2	25	occupied
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8	30	occupied
9		free
10	10	occupied
11	23	occupied
12	12	occupied

+ {25, 30}

Linear Probing

```
// This method is only used if there is at least a
// free slot in the table: _n < _M
template <typename Key, typename Value,
          template <typename> class HashFunct>
void
  Dictionary<Key,Value,HashFunct>::insert_linear_probing(
     const Key& k, const Value& v) {

   int i = hash(k);
   while (not _Thash[i]._free and _Thash[i]._k != k)
      i = (i + 1) % _M;
   }
   _Thash[i]._k = k; _Thash[i]._v = v;
   if (_Thash[i]._free) ++_n;
   _Thash[i]._free = false;
}
```

Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup(
       const Key& k,
       bool& exists, Value& v) const {

   int i = lookup_linear_probing(k);

   if (not _Thash[i]._free and _Thash[i]._k == k) {
      exists = true;
      v = _Thash[i]. _v;
   }
   else
      exists = false;
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup_linear_probing(
       const Key& k) const  {

   int i = hash(k);
   int visited = 0;  // this is only necessary if
                     // _n == _M, otherwise there is at least
                     // a free position

   while (not _Thash[i]._free and _Thash[i]._k != k
          and visited < _M) {
      ++visited;
      i = (i + 1) % _M;
   }
   return i;
}
```

Deletions in Open Addressing

There is no general solution for true deletions in open addressing tables. It is not enough to mark the position of the element to be removed as “free”, since later searches might report as not present some element which is stored in the table.

The general technique that can be used is lazy deletions. Each slot can be free, occupied or **deleted**. Deleted slots can be used to store there a new element, but they are not free and searches must pass them over and continue.

Deletions in Linear Probing

For linear probing, we can do true deletions. The deletion algorithm must continue probing the positions after the removed element, and moving to the emptied slot any element whose hash address is equal (or smaller in the cyclic order) to the address of the emptied slot. Moving an element creates a new emptied slot, and the procedure is repeated until an empty slot is found. In our implementation we will use the function $\texttt{displ}(j,i)$ which gives us the distance between j e i in the cyclic order: if $j > i$ we must turn around position _$M-1$ and go back to position 0.

```
int displ(j, i, M) {
  if (i >= j)
     return i - j;
  else
     return M + (i - j);
}
```

Linear Probing

```
// we assume _n < _M

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::remove_linear_probing(
       const Key& k) const  {

    int i = lookup_linear_probing(k);

    if (not _Thash[i]._free) {
       // _Thash[i]is the element to remove
       int free = i; i = (i + 1) % _M; int d = 1;
       while (not _Thash[i]._free) {
             int i_home = hash(_Thash[i]._k);
             if (displ(i_home, i, _M) >= d) {
                _Thash[free] = _Thash[i]; free = i; d = 0;
             }
             i = (i + 1) % _M;  ++d;
       }
       _Thash[free]._free = true; --_n;
    }
}
```

The Cost of Linear Probing

Besides simplicity, linear probing offers several advantadges since synonyms tend to group into consecutive positions; this is particularly useful when the hash table is stored in external memory, in combination with some bucketing technique: each slot of the table stores up to B elements.

On the other hand linear probing suffers the problem of clustering and its performance rapidly degrades as the loading factor $\alpha = n/M$ approaches 1.

Linear Probing

For $\alpha < 1$ the cost of successful searches (and of modifications of the value associated to a key) is proportional to

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right),$$

whereas the cost of unsuccessful searches, insertions and deletions is proportional to

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right).$$

The Cost of Linear Probing

The phenomenon called clustering appears in open addressing strategies when an element does not occupy its “preferred” location due to the presence of some element which is not a synonym (so it is not the “preferred” location of that element either).
As more and more new elements are inserted and the load factor $\alpha \to 1$, groups of synonyms merge into big clusters; when we lookup for some key we have not only to examine its synonyms but perhaps some large number of unrelated elements.

Rehashing

Many programming languages allow the creation of arrays giving their size in execution time. Then we can use resizing to avoid high load factors. When a new insertion makes the load factor larger than some fiexd threshold, a new array that roughly doubles the size of the current array is created (claiming the storage from dynamic memory). The contents of the old array are inserted (“rehash”) into the new array, and the memory used for the old array is freed (released back to the dynamic memory).

Rehashing

Rehashing is a costly operation, with time proportional to n. But we need to do it only from time to time, the frequency exponentially decaying with the size of the table. Rehashing allows us to grow a dictionary without imposing a maximum number of elements, while keeping a reasonable load factor. The technique can be used in reverse, to avoid very low load factors (which mean a substantial waste of memory).

C++ `vector`s have the `resize` method, that does exactly this (allocate new array, copy contents, release old array). It can be used to implement hash tables, but we will have to take care of the rehashing anyway.

Rehashing

It can be proved that, although a single insertion/deletion might have expected cost $\Theta(n)$ (because of the rehashing), any sequence of n operations will have expected total cost $\mathcal{O}(n)$. We can say then that the n operations have expected amortized cost $\mathcal{O}(1)$ each.