Compilers

Spring term

Alfonso Ortega: alfonso.ortega@uam.es
Enrique Alfonseca: enrique.alfonseca@uam.es

Chapter 2: Symbols Table

Lectures 2 and 3

Search Methods Reminder

Linear search

- **MTP2** and **AA**: different search methods and their complexity.
- **Time complexity**: number of basic operations that have to be performed: number of comparisons that have to be done in a table with n elements until we find the one we want..
- **Linear search**
  - Worse time complexity (max. number of data comparisons):

$$W_{LinearSearch}(n)=n$$

  - Average time (mean number of data comparisons):

(assuming equiprobability)

$$A^{e}_{LinearSearch}(n)=\sum_{i=1}^{n} i.p(k=T[i])=\frac{1}{n}\sum_{i=1}^{n} i$$

Search Methods Reminder

Linear Search

```
index LinearSearch(table T, index P, index U, key k)
      For i from P to U:
            If T[i] == k:
                  return i;
      return "error";
```

Search methods reminder

Binary search

- **Binary search (tables with ordered information)**
  - Average time (mean number of comparisons required to find a data item in a table with n elements):

$$A_{BinarySearch}(n) \sim \log_2(n)$$

```
index BinarySearch
 (table T, index P, index U, key K)
      while P≤U
            M=⌊(P+U)/2⌋
            If T[M] < K
                  P=M+1;
            else If T[M]>K
                  U=M-1;
            else return M;
        return Error;
```

Compilers

5

Dictionary data structure Reminder

Definition

- The **dictionary** data structure is:
  - A data structure where items are ordered.
  - It allows the following operations: **search, insert, delete**

```
Position Search(key k, dictionary D)
```

  - where
    - k is the key looked for.
    - D is the dictionary,
  - Provides the position of the key; or a predefined value (e.g. -1) if it is not present in the dictionary.

Compilers

6

Dictionary data structure Reminder

Dictionary data structure and operations

```
error_code Insert (key k, dictionary D)
```

- where
  - k is the key that will be inserted in the dictionary.
  - D is the dictionary where k will be inserted (it will be modified during this operation)
- The return code indicates whether the key has been inserted correctly:
  - Search k in D
  - If not present, introduce k inside D, so it is placed orderedly w.r.t. the remaining items.

Dictionary Data Structure reminder

Dictionary Data Structure and Operations

```
error_code Delete(key k, dictionary D)
```

- where
  - k is the key that will be deleted.
  - D is the dictionary from which k will be deleted.
- The return code indicates whether the deletion could be performed correctly. The operation performed is:
  - Look for k inside D.
  - If present, readjust D so k disappears, and the remaining items are still ordered.

Dictionary Data Structure Reminder

Implementation with ordered vectors

- It is possible to implement a Dictionary with the data structure “Ordered Vector”.
- Ordered vectors are collections of elements, in which one can access each element by knowing its position, and the placement of the elements is done according to an order.
- In the following, we shall study how to code the operations, and the space and time requirements.

Dictionary data structure Reminder

Search on ordered vectors

- Search is the most costly basic operation (for either search, insert or delete) and, therefore, it affects greatly the performance of the data structure.
- Remember that

$$A_{BinarySearch}(n) \sim \log_2(n)$$

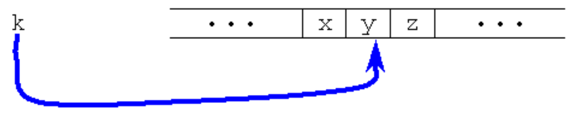
- This, for algorithms based on comparing keys, is an acceptable performance.

- The purpose here is to analyse how the other operations (insert and delete) could be performed, to ensure that it will not spoil the overall performance.

Dictionary data structure Reminder

Insertion in Ordered Vectors

- We can imagine insertion in the following way:
  1. Look for the data k to be inserted in the dictionary D.
     - The search will find the position where k should be, if not already there.



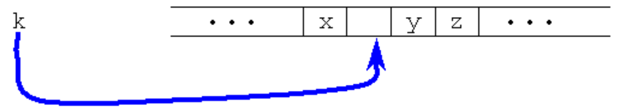
The expected average cost for this search will be:

O(log(n))

Dictionary data structure Reminder

Insertion in Ordered Vectors

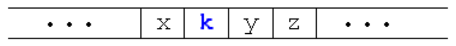
2. Make place for this new item. A possibility is to have to move all the other items one position behind.



Therefore, we can consider that the performance in execution time will depend linearly on the size of the table.

O(n)

3. Finally, we shall write item “k” in the corresponding position in D.



This last step will require a constant time:

O(1)

Dictionary data structure Reminder

Insertion in Ordered Vectors: conclusions

- The step with the worse complexity is step 2, O(n), which is the one that actually inserts the new element in its position.
- It can be seen that the overall insertion time performance with ordered vectors is:

$$O(n)$$

- The time complexity, compared to search's `O(log(n))`, is rather large. This is enough to **reject ordered vectors as a data structure** for the Dictionary.

Dictionary data structure Reminder

Implementation of the dictionary as ordered binary trees

- Second possibility: implementation as ordered binary trees
- We can define binary ordered trees in the following way:
  - Let T be the tree
    - key (T) the key associated to T's root.
    - left (T) the sub-tree which is the left child of T.
    - right (T) the sub-tree which is the right child of T.
  - T will be an ordered binary tree if and only if:
    - $\forall$ T' node of T,
      key (left (T')) $\leq$ key (T') $\leq$ key (right (T'))

Dictionary data structure Reminder

Search in ordered binary trees

- The basic operation of search can be implemented with a recursive procedure:

```
BinTree Search(key k, BinTree T)
   If empty(T) return empty_tree
   If k == key(T) return T
   If k < key(T)
      return(Search(k,left(T));
   If k > key(T)
      return(Search(k,right(T));
```

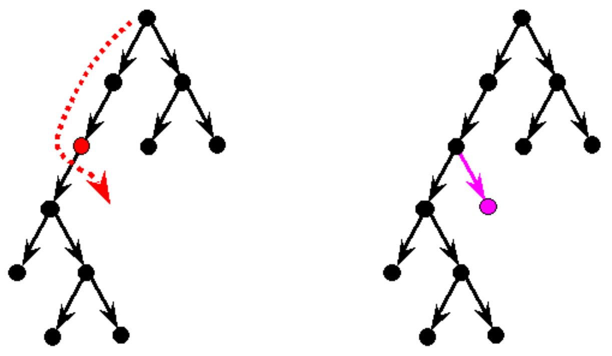
- The time complexity of this algorithm is related to the depth of the tree:

$$n_{\mathrm{Search}}(k,T)=O(\mathrm{depth}(T))$$

Dictionary data type Reminder

Insertion inside ordered binary trees

- We can represent the insertion of a new item in an ordered binary tree in the following way:
  - Assuming that the new data item will be placed in the location indicated by the red arrow...



Dictionary data type Reminder

Insertion inside ordered binary trees

- Let's assume that we modify the code of the search so that, if the element we're looking for is not in the tree, we return the node that should have been its father. For instance, in the previous graph, if we look for the pink node and it is not there, we would get the red node instead.
- The following pseudo-code would code the insertion operation:

```
state Insert(key k, BinTree T)
   BinTree T', T'';

   T'  = Search(*)(k,T);
   T'' = new_node(k);
   <If the new node could not be created, return error code>
   If k < key(T') left(T')=T'';
   else right(T')=T'';
   return "ok"
```

Dictionary data type Reminder

Insertion inside ordered binary trees

- In this function, most of the work again will be devoted to searching. Therefore, it is easy to imagine that the average performance also depends on the depth of the tree:

$$n_{Insert}(k,T)=O(depth(T))$$

Dictionary data type Reminder

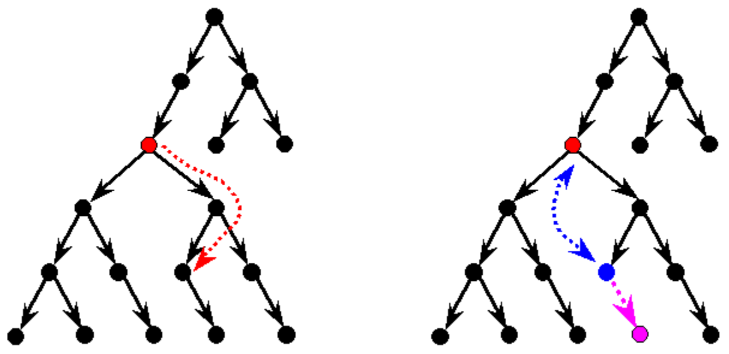
Deleting inside an ordered binary tree

- To delete from an ordered binary tree...
  - Given any node, the next node in the tree (following the relation of order in the tree) can be found with the following procedure:
    - Go down a level to the right child, and next
    - Go down as down as possible along the left children.
  - This is because the node following a given node must be the lowest of all the nodes that are higher than it. In other words, the left-most child of the right sub-tree.
- It is easy to check that, if we remove a node, we can reconstruct the binary tree if:
  - We substitute the node deleted with his successor.
  - We re-structure the descendants of the successor node.

Dictionary data type Reminder

Deletion in ordered binary trees: example

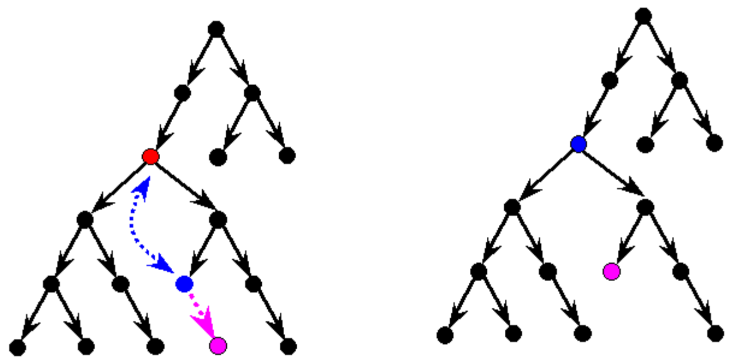
- The following illustrates graphically the previous procedure:
  - We plan to remove the red node.
  - The successor will be the blue node.



Dictionary data type Reminder

Deletion in ordered binary trees: example

- The node to be deleted is replaced by its successor.
- The descendants of the successor (in the example there is just one, the pink node) will be replaced in the tree.



Compilers 21

Dictionary data type Reminder

Deletion in ordered binary trees: example

- The following pseudocode performs this operation.
- The function "DeleteAndReadjust" must look for the successor, interchange it with "k", and re-structure a (small) subset of the nodes.

```
error_code Delete(key k, BinTree T)
   BinTree T';

   T'=Search(k,T);
   <If not in the tree, return error code>
   DeleteAndReadjust(T',T);
```

Compilers 22

Dictionary data type Reminder

Deletion in ordered binary trees: example

- The work of deletion can be summarised in the following steps:
  - Search for two nodes (the one deleted, and its successor). Each of the searches has a complexity of `O(depth(T))`
  - Replace a small set of nodes, a task which we might consider constant `O(1)`
- Thus, the overall complexity will be

  `O(depth(T))`
- In the average case, the complexity is the following:

$$A^{e}_{Depth}(n) = 2.\log(n) + O(1)$$

Dictionary data type Reminder

Ordered binary trees: conclusions (I)

- In the average case, the three operations (search, insert, delete) will have a complexity of:

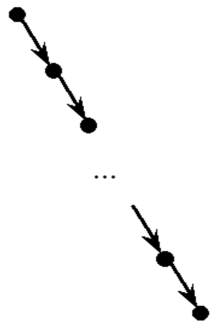
  `O(log(n))`
- It is important to point out that the performance is really dependent on the depth of the tree.
  - If the tree is extremely ill-suited to the task, it may become rather slow.

Dictionary data type Reminder

Ordered binary trees: conclusions (II)

- Remember the following case:
  - Creating an ordered binary tree for the following sequence of items:

    `1,2,...,n`
  - Because the items are pre-ordered, the tree obtained will be as follows:



  - This is really an ordered linked list.
  - Therefore, the overall time performance will be

    `O(n)`

  as with ordered vectors.
- It is not acceptable to allow for the possibility of these trees.

Dictionary data type Reminder

Implementation as an AVL

- To improve the previous situation, Adelson-Velskii and Landis proposed a kind of trees in which the previous situation could not happen. The complexity, even in the worst case, is `O(log(n))`.
- Their idea is to rearrange the nodes inside the trees, so we always obtain an ordered binary tree in which no sub-tree is too unbalanced. So, it is only allowed to have one-level unbalances, i.e., the depth of the right and the left subtrees may differ in at most 1.
- This was obtained with a couple of “rotation” operations of the tree during the insertion of new elements.

Dictionary data type Reminder

Conclusions

- Using data structures based on key-comparisons for searching, the best performance obtained is

  `O(log(n))`

- How to obtain better performances?
- Is it possible to obtain constant performances in search, using any other kind of procedure?

Compilers

27

Dictionary implementation with Hash Tables

Hash tables

- The idea motivating hash tables is to **associate a unique position in a table to each possible value** that we might possibly insert in the dictionary:
  - **Search** would consist in **obtaining the position** associated to that key, and immediately **accessing that position** to retrieve the information from the table. The **"cost" would not depend on the table size.**
  - **Insertion** would first check that the corresponding **position** was empty, and would next access it to insert the item. Again, the **"cost" would not depend on the table or the item's size.**
  - **Deletion** would consist in **accessing the position** associated to the key, to remove the data in that position. Once more, the **"cost" would not depend on the table or the item's size.**
- This is why these tables are called **calculated-entry table**, because the position to access the table is calculated by means of a function.

- An important concept is the **load factor** (λ). If there are n items inserted in a hash table of size m, this factor is defined as:

$$\lambda = \frac{n}{m}$$

Compilers

28

Dictionary implementation with Hash Tables

Hash function

- To calculate the position in a table, associated to each value:
  - Choose a part of the item that will be considered its **key**.
  - Code a **function** h that takes as input that key, and calculates the table position.

  - If we have a set K of keys, and the positions in the table range from 0 to m, then h is defined as::

$$h: K \to [0,m]$$
$$k \to h(k)$$

- This function should, ideally, be an injective function. In this way, we'll ensure that two keys which are different will receive different positions in the table.

$$\forall k,k' \in K \; (\; k \neq k' \;\; \Rightarrow \;\; h(k) \neq h(k') \;)$$

Dictionary implementation with Hash Tables

Hash functions which are not injective: collisions

- It would be too costly to work with injective functions.
- Therefore, there may be cases in which two different keys correspond to the same position in the table:

A **collision** happens when two keys which are different are assigned the same table position

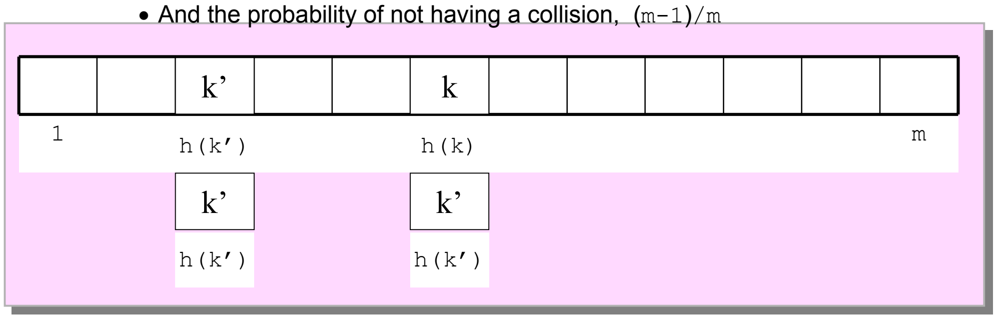
- Formally,

$$\exists \; k,k' \in K \mid (\; k \neq k' \;\; \wedge \;\; h(k) = h(k') \;)$$

Dictionary implementation with Hash Tables

Hash functions which are not injective

- As collisions are allowed, the emphasis will be placed in minimising the number of times they occur:
  - This can be understood as the fact that
    - The probability of collision is small, and
    - The probability of not having a collision is large.
  - If a table has `m` positions, k is already placed in the table, and we insert k',
    - The minimum probability of collision is `1/m`
    - And the probability of not having a collision, `(m-1)/m`



`p(¬colisión)=(m-1)/m` `1/m`

31

Dictionary implementation with Hash Tables

Hash functions which are not injective: pseudo-random functions

- The previous scenery could be obtained with **a pseudo-random hash function**:
  - Each time we **insert a key k,**
  - The program "**throws a dice**" with **`m` sides** and obtains a **result `i`**
  - **`h(k)=i`**
- This, although produces the probability desired, however, does not provide a **realistic method to retrieve** the information afterwards, from **k**:
  - We throw the dice with **`m` sides** and obtain a different? **result** `j`
  - **`h(k)=j`**
- Given the design of the experiment, it is not sure that, given the same key, we shall obtain the same position in the table both for inserting and retrieving the contents.
- **It is not possible to** implement this procedure for practical usage, because we won't be able to retrieve the inserted information.

32

Dictionary implementation with Hash Tables

Hash functions which are not injective: multiplication hash function

- A first alternative is to use hash functions that distribute the keys in a more-or-less uniform way through the table.
- **MTP2**:
  - Multiplication hash functions:
    - The product of the size of the table (m) times a uniform function with image [0,1] produces a real function with image [0,m], which can be casted to integer to obtain h.
    - For instance,

Let's take a number $\phi \in R-Q$ (irrational)

We can use the function "fractionary part of x", defined as follows:

$(\cdot):R\rightarrow[0,1) \mid (x)=x-\lfloor x\rfloor$

A possible hash function is

$h(k)=\lfloor m(k\phi)\rfloor$, where, in general,

$$\phi=\frac{\sqrt{5}-1}{2}, m=2^{p}, p \text{ primo}$$

Dictionary implementation with Hash Tables

Hash functions which are not injective: multiplication hash function

- Let's illustrate the function with the following example: m=25 y $\phi=\pi$:

k	$k\pi$	$h(k)$
1	3.141592654	3
2	6.283185307	7
3	9.424777961	10
4	12.56637061	14
5	15.70796327	17
6	18.84955592	21
7	21.99114858	24
8	25.13274123	3
9	28.27433388	6
10	31.41592654	10

k	$k\pi$	$h(k)$
11	34.55751919	13
12	37.69911184	17
13	40.8407045	21
14	43.98229715	24
15	47.1238898	3
16	50.26548246	6
17	53.40707511	10
18	56.54866776	13
19	59.69026042	17
20	62.83185307	20

Dictionary implementation with Hash Tables

Hash functions which are not injective: division hash function

- **MTP2**:
  - **Division hash functions**:
    - The “module” function is used to obtain an integer value from the key.
    - For instance,

Table size: `m=|T| prime`

We can define `h(k)= k%m`

Compilers

35

Dictionary implementation with Hash Tables

Hash functions which are not injective: division hash function

- Examples, with `m=13` :

k	*h(k)*
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	0

k	*h(k)*
14	1
15	2
16	3
17	4
18	5
19	6
20	7
21	8
22	9
23	10
24	11
25	12
26	0

k	*h(k)*
27	1
28	2
29	3
30	4
31	5
32	6
33	7
34	8
35	9
36	10
37	11
38	12
39	0

Compilers

36

Dictionary implementation with Hash Tables

Other hash functions

- There are many other hash functions used in practice:
  - Obtain a numeric value `w` from an identifier,
    - Arithmetic sum of the ASCII codes of the letters in the identifier.
    - XOR of the binary representation of the letters in the identifier.
  - Transform `w` into a value which is valid for the hash table:
    - With integer arithmetic (e.g. module)
    - With operations on the binary representation:
      - In this case, if we have a hash table with 2^p entries (p is usually a prime number), we need p bits to represent a position in the table.
      - A possibility is to use `p` bits taken from the binary representation of w, or from any operation performed with w:
        - » `w×w`
        - » The last `p` bits,
        - » etc...

Compilers 37

Dictionary implementation with Hash Tables

Solving collisions

- The problem with hash tables appears with collisions.
- There are several methods to solve them:
  - Open address hash functions
    - With random rehash
    - With linear rehash
    - With multiplication rehash
    - With quadratic rehash
  - Linked hash functions:
    - Internal
    - With overflow

Compilers 38

Open address hash

Concepts

- The data items end up all scattered along the table; they do not form contiguous clusters.
- Collisions are solved **inside the table**.
- Therefore, **the size of the table** should always be **larger** than the number of data items, so as to have, always, empty places.
- If the position provided by the hash function is occupied, then a special procedure will be used to scan other positions until we find an empty place.
- In this way, the position of any item depends on
  - Its key
  - The state of the table when it was inserted

  In other words, the key alone is not enough to know the position of the item, which remains "open". That's the reason for the name of these tables.
- The procedure for placing a key whose position was previously occupied is called "**rehash**".
- The main aims are: (a) keep a good performance; (b) Be able to fill as many places in the table as possible.

Open address hash

Search

- All the variants of this method share the same search pseudo-code:

```
index Search(key k, HashTable T)
   index position = hash_function(k,T), initial=position;

   If k == T.data[position].key return position;
   else
   {
      while not empty(T.data[position])
            and not already_visited(position)
            and not k == T.data[position].key
      {position = (position + delta(i++, initial))mod size(T);}
      if empty(T.data[position]) return -1; /*Not there*/
      if already_visited(position) return -1;
            /* The table does not have the value, nor free space */
      if k == T.data[position].key return position;/*Found!*/
   }
```

Open address hash

Insertion

- All the variants of this method share the same insertion pseudo-code:

```
index Insert(key k, HashTable T)
    index position = hash_function(k,T), initial = position;
    int i=0; /* Number of retries */
    If k == T.data[position].key return position; /* Already there */
    else{
        While not empty(T.data[position])
                and not already_visited(position)
                and not k == T.data[position].key
        {position = (position + delta(i++, initial))mod size(T);}
        if empty(T.data[position]){/* It was not there: insert it */
                T.data[position].key = k;
                return position;}
        if already_visited(position) return -1;
                /* There is not empty space in the table */
        if k == T.data[position].key return position;
        /* It was already there */
    }
```

Open address hash

Observations

- When there are collisions, the search is attempted again by adding a number obtained through the call to **`delta(i++)`**:
- The different variants of open-address hash differ in the implementation of this function.
- Consider that there are cases in which we consider that **there is not free space in the table:**
  - When the table is completely full
  - When the series of positions that have been tried visit a position that has been tried previously. In this case, even though there might be free spaces in the table, we shall never be able to find them using the rehash function.

Open address hash

Linear rehash

- In the case of linear rehash, the rehash function is:

```
int delta(int number_of_retry, int initial)
{
    return (initial+number_of_retry);
}
```

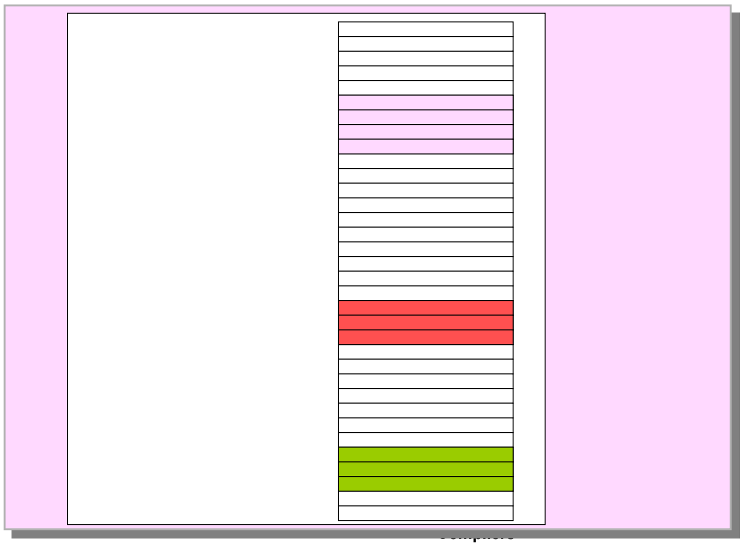
- The series of positions will be:

```
position
position+1
position+2
...
position+i
```

Open address hash

Linear rehash

- When there are collisions, data with the same values for the hash function will appear grouped in the table, in consecutive positions:



Open address hash

Linear rehash: performance

- It can be proven that:

$$A^f_{LinearOpenHash}(n,m) \cong \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

$$A^s_{LinearOpenHash}(n,m) \cong \frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$$

- This conclusion can also be found in other sources
  - Peterson: obtained by simulations
  - Schay y Spruth: assuming that all keys that go to position `i` will be inserted before the keys that go to position `i+1`.

Open address hash

Multiplicative rehash

- In the case of multiplicative rehash:

```
int delta(int number_retry, index initial)
{
    return (initial*number_retry);
}
```

- The series of positions here will be the following:

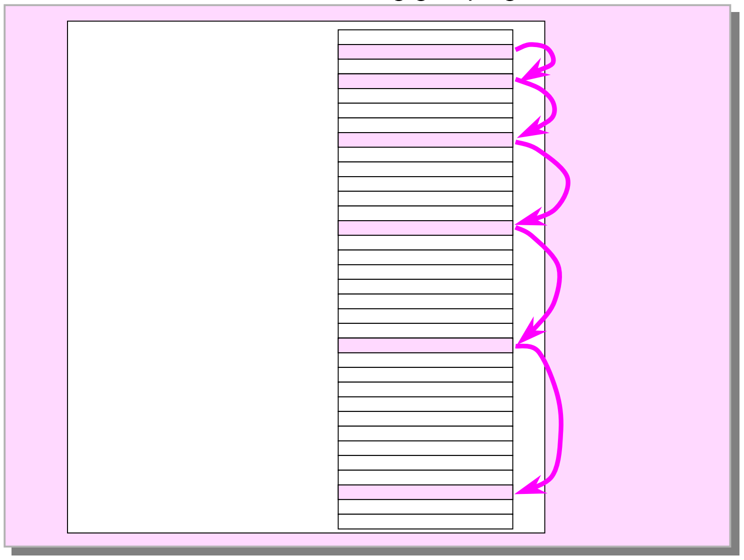
```
position
2*position
...
i*position
```

- Observations:
  - The initial position cannot be zero (otherwise, all the rest would be zero as well)
  - If the size of the table is a prime number, it will cover the whole table.

Open address hash

Multiplicative rehash

- When there are collisions, the data items with the same hash value will be scattered in the table, avoiding groupings.



47

Open address hash

Quadratic rehash

- In the case of quadratic rehash, the rehash function would be the following:

```
int delta(int number_of_retries, index initial)
{
    return initial +
            (a*number_of_retries² +
             b*number_of_retries + c);
}
```

48

Open address hash

Random rehash

- In the case of the random rehash, the function is the following:

```
int delta( )
{
    return ( random() );
}
```

- As already said, this is not usable in practice, as it would be impossible to retrieve the items from the table.
- It may be interesting, as it may allow us to perform more easily an analysis of the average-case and the worst-case complexity of the algorithms.

Open address hash

Random rehash: performance

- It can be proven that:

$$A^{s}_{RandomOpenHash}(n,m) \cong \frac{1}{\lambda}\log\left(\frac{1}{1-\lambda}\right)$$

$$A^{f}_{RandomOpenHash}(n,m) \cong \frac{1}{1-\lambda}$$

Open address hash

Resizing the table

- It may be adequate to resize the table in the following cases:
  - When the table is full
  - When the table does not provide empty places for a given key.
  - When the performance goes down below a certain threshold.
- The result of the resize will be:
  - A larger table.
  - All the values from the previous table will be inserted in this larger table.
- The new size of the table must be coherent with the design decisions taken.

Linked Hash tables

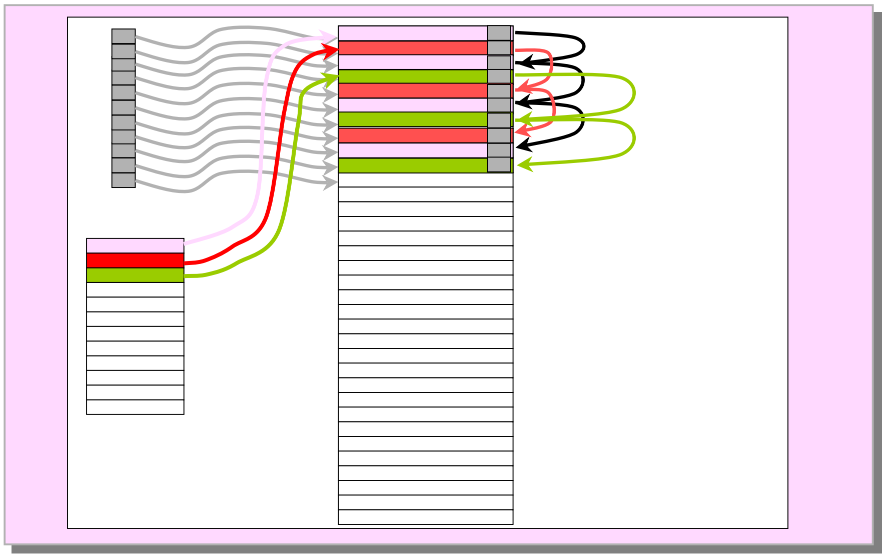
Concepts

- In a **linked hash table**, each position of the table contains a linked list with all the items that collisioned in that position.
- In this way, the number of items in the table may be larger than the number of positions.
- It is usually implemented in the following two ways:
  - Internal linking.
  - Overflow linking.

Linked Hash tables

Internal linking: Insertion

- Each position in the hash table contains a linked list with all the elements that collisioned.

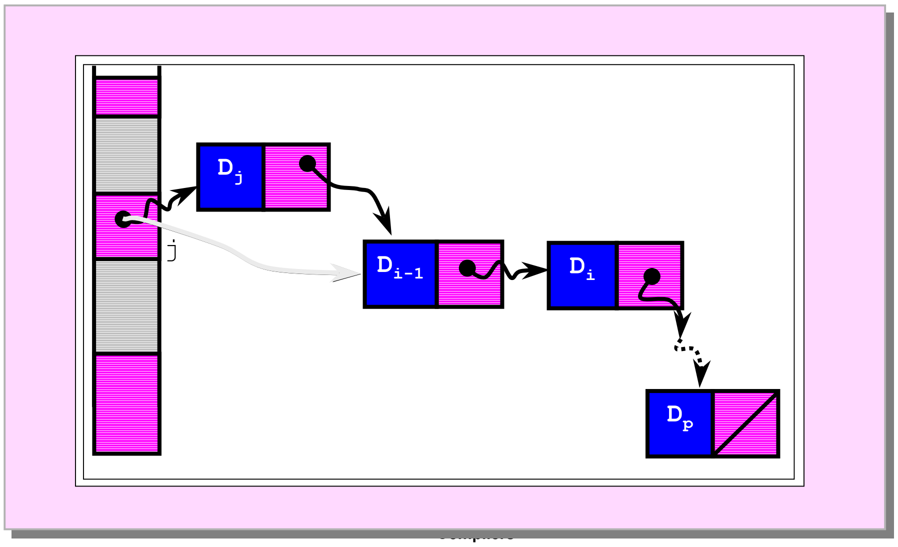


53

Linked Hash tables

Overflow linking: insertion

- Each position in the hash table contains a linked list with all the elements that collisioned there.



54

Linked Hash tables

Search

- In any of the two implementations, search is performed in the following way:
  - Calculate the hash value corresponding to the key.
  - Look (e.g. with linear search) in the linked list associated to that hash value.

Compilers

55

Linked Hash tables

Performance

- It can be proven that:

$$A^{f}_{Search}(n,m) = \frac{n}{m} = \lambda$$

$$A^{s}_{Search}(n,m) = 1 + \frac{1}{2}\lambda - \frac{1}{2m}$$

Compilers

56

Linked Hash tables

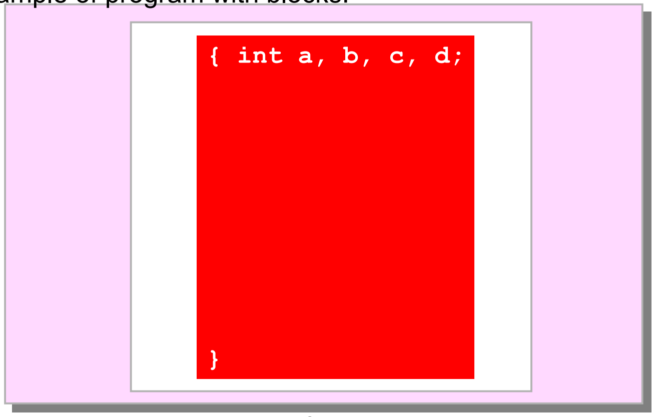
Resizing the table

- It will be appropriate to resize the table when the performance goes down below a certain threshold.
- The resize will imply:
  - Creating a larger table.
  - Reintroducing all the keys from the previous table.
- As before, the new size must be coherent with the design decisions taken.

Symbol tables with block structures

Concepts

- **Programming languages with block structures**:
  - Those languages in which some identifiers only exist inside some sections of the code, and not in others.
  - For instance, Algol, PL/I, C, Java, Pascal…
  - Example of program with blocks:



```
{ int a, b, c, d;



}
```

Symbol tables with block structures

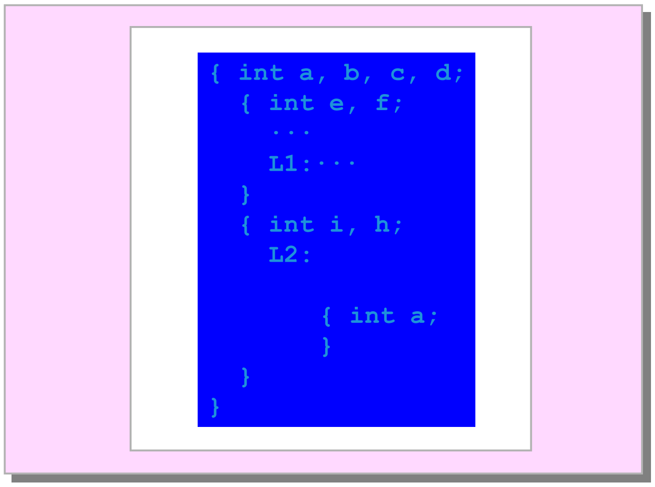
Concepts

- **Scopes associated to each line of code:**
  - Every line of code is inside a set of blocks or scopes.
  - This defines the scopes or blocks in which each identifier is valid.
- **Current scope**:
  - It is the scope defined by the deepest block in which the current code line is.
- **Open scopes**:
  - All the scopes that include a line of code (directly, or through other inner scopes) are said to be open with respect to that line.
- **Closed scopes**:
  - All the scopes that do not include a line of code (directly, or indirectly) are said to be closed with respect to that line.

Symbol tables with block structures

Concepts

- **Example**



```
{ int a, b, c, d;
  { int e, f;
    ...
    L1:...
  }
  { int i, h;
    L2:

       { int a;
       }
  }
}
```

Symbol tables with block structures

Concepts

- **Rules to determine which identifiers are active:**
  - The only active identifiers are:
    - Those defined in the current scope.
    - Those defined in open scopes which include the current one.
  - Whenever there are two entities with the same name, in different scopes, the only one accessible will be **the one defined in the deeper scope**.
  - The names of the **arguments of a procedure** are local to that procedure, and they are not accessible from outside.
  - The **name of a procedure**, in order to be accessible, must be:
    - Local to the block where it was defined.
    - Global in the program.

Symbol tables with block structures

A table per scope, one pass

- **General structure**
  - A list of scopes is created so that each has a hash table.
  - The open scopes are in the list, in the inverse order of the opening time.
  - The current scope, then, is in the first place (the last one to be opened).
  - Once a scope is closed, in one-pass compilers, they are never opened again, so all the resources associated with it can be freed.
  - Therefore, in one-pass compilers, the list of open scopes can be implemented as a stack.
- **Insertion algorithm:**

```
status insert(id key, ScopesHashTable T)
{
   if exists(key, current_scope(T) return FAIL;
   insert_key( current_scope(T) );
}
```

Symbol tables with block structures

A table per scope, one pass

- **Search algorithm:**

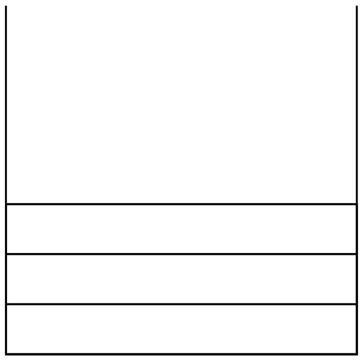
```
Hash_entry search(id key, ScopesHashTable T)
{
    HashTable aux_t;

    aux_t = current_scope(T);
    while (not empty_hash_table(aux_t))
    {
            If exists(key, aux_t) return aux_t;
            aux_t = next_table(aux_t,
                                   TablaHashAmbitos);
    }
    return NULL;
}
```

Symbol tables with block structures

A table per scope, one pass: example

```
{ int a, b, c, d;
  { int e, f;
    ...
    L1:...
  }
  { int i, h;
    L2:
        { int a;
        }
    L3:
  }
}
```



Symbol tables with block structures

One table per scope, more than one pass

- **General structure: differences with respect to one-pass compilers:**
  - In this case, the information about closed scopes should be kept, because the compiler might need to perform another pass over that scope.
  - When a block is opened, its information is put before the information of its parent block.
  - When a block is closed, e.g. with a closing brace }, rather than freeing the top element in a stack of hash tables, the hash table for that scope is kept, but marked as closed.
  - In this way, the current block is always the first one after the closed blocks.
  - All the closed blocks will be the ones grouped before the open ones.

Compilers

65

Symbol tables with block structures

One table per scope, more than one pass: example

- **Exercise**
  - Write the sequence of contents in the hash table (one per scope) of a compiler with more than one pass, for the following fragment of code:

```
{ int a, b, c, d;
  { int e, f;
    ...
    L1:...
  }
  { int i, h;
    L2:
        { int a;
        }
    L3:
  }
}
```

Compilers

66

Symbol tables with block structures

One table per scope: discussion

- **Problems**
  - If we have more than one table, that may fragment in excess the space dedicated to storing hash tables.
  - The fact that we have to search in several tables may worsen the performance.

Symbol tables with block structures

Just one table

- In this case, all the identifiers will be inserted in the same table.
- Each identifier will have information about the scope to which it belongs.
- **Possible implementation:**
  - Using
    - An array for the data about the blocks.
    - Other for the identifiers. It may be just one structure that uses the end for the identifiers of the open blocks, and the beginning for the ids. of the closed ones. On the other hand, there might be two: one for the open scopes and one for the closed scopes.
  - Each block will contain the following information:
    - Block number.
    - Number of the parent block.
    - Number of identifiers in this block.
    - Pointer to the information of the identifiers in this block.

Symbol tables with block structures

Just one table

- **Block creation**
  - When a new block starts, {
    - Its information is added to the array of blocks:
      - It is assigned a new number, e.g. 1+the number of the last block.
      - Its parent block is the current block.
      - Initially, it will have 0 identifiers.
      - The pointer to the structure with its identifiers will be initialised. Initially, it will contain the ids. contained by its parent.
- **Declaration of a new identifier**
  - Check: that the new identifier has not been already defined in the current block.
  - The number of ids. in the current block is incremented in 1 unit.
  - The pointer to the identifiers in the block is updated, so it points to the new identifier.
  - The information about the new identifier is added to the zone of open identifiers.

Symbol tables with block structures

Just one table

- **Block closure**
  - When we find the symbol closing the current block, }
    - Its identifiers are removed from the zone of open ids., and they are copied to the zone of the closed ids.
    - The block is closed, and the current block is now its parent block.
- **Search of an identifier**
  - It is looked for in the hash table.
  - If there are more than one, we'll take the last one to be defined (i.e. the one defined in the innermost block).

Symbol tables with block structures

Just one table: example

- **Example**
  - The same example as before:

```
{ int a, b, c, d;
  { int e, f;
    ...
    L1:...
  }
  { int i, h;
    L2:
        { int a;
        }
    L3:
  }
}
```

Symbol tables with block structures

Just one table: evaluation

- **Advantages**
  - The chances that we shall not use too much space in vain are lower than in the method with one table per scope.
  - The search in just one table may have better performance.

Treatment of identifiers

Key and value

- In each entry in the symbols table the compiler will store all the information needed.
- This uses to be the following:
  - **Identifier class:**
    - Variable
    - Function, procedure
    - Label
    - Enumeration value, etc.
  - **Kind:**
    - Integer
    - Real
    - Boolean
    - Complex number
    - Character
    - String
    - Structure
    - Declared type
    - Operator
    - Return value (or void) if function.
    - Number of arguments if function

Treatment of identifiers

Key and value

- **Precision, scale**
- **Shape:**
  - Scalar
  - Vector
  - Array
  - List

- If function argument
- If it has already been initialised
- If it is a recursive function
- List of parameters, local variables, etc.