Searching
— T.06 —

Summary:

- Searching unordered data (in an array, in a linked list, in a binary tree, or in a hash table)
- How to improve the search time (data reordering)
- Searching ordered data (in an array — binary search — or in an ordered binary tree)
- Exercises

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Searching unordered data (part 1, array and linked list)

Searching for a certain piece of information when that information is not organized may require a complete pass over all data. That is an $O(n)$ operation. (If the information is stored in a hash table of appropriate size, searching for a given piece of information is, on average, a $O(1)$ operation. This is descriped in detail in the T.05 slides.)

On an array, that can be done in as follows (the function returns the **first** index `i` for which `data[i] == value`, or -1 if none exists):

```
int find(T *data,int data_size,T value)
{
  int i;

  for(i = 0;i < data_size && data[i] != value;i++)
    ;
  return (i < data_size) ? i : -1;
}
```

On a linked list, it can be done in the following way (the function returns a pointer to the **first** node `n` for which `n->data == value`, or NULL if none exists):

```
node *find(node *head,T value)
{
  while(head != NULL && head->data != value)
    head = head->next;
  return head;
}
```

Searching unordered data (part 2, binary tree and hash table)

On an (unordered) binary tree, searching for a node n for which n->data == value can be done in the following way (note the use of recursion):

```
node *find(node *n,T value)
{
  node *nn;

  if(n == NULL || n->data == value)
    return n;
  if(n->left != NULL && (nn = find(n->left,value)) != NULL)
    return nn;
  if(n->right != NULL && (nn = find(n->right,value)) != NULL)
    return nn;
  return NULL;
}
```

If this function does not return NULL then it returns a pointer to **one** node that satisfies n->data == value (in an unordered tree, the concept of **first** is not well defined).

[**Homework:** It more than one tree node with the same value exist, which one is returned by this function?]

Of course, searching for unordered information in a hash table can be done in an efficient way. After all, the hash function will restrict the search to all entries that collide with the entry where the information we are looking for is located.

How to improve the search time

For unordered data, the search time can be improved if the number of queries for some data items is not uniformly distributed (check the example near the end of the T.04 lecture).

One possible way to improve the average search time consists of reordering the information after each search, so that the data item that was found becomes closer to the beginning of the relevant data structure. For arrays its index becomes smaller, for linked lists its node becomes closer to the head of the list, and for binary trees its node becomes closer to the root of the tree.

The following function does this for an array:

```
int self_optimizing_find(T *data,int data_size,T value)
{
  int i;

  for(i = 0;i < data_size && data[i] != value;i++)
    ;
  if(i > 0 && i < data_size)
  {
    T tmp = data[i];
    data[i] = data[i - 1];
    data[--i] = tmp;
  }
  return (i < data_size) ? i : -1;
}
```

Other optimizations are possible. For example, if the same query has a tendency to be performed two or more times in a row, it will be advantageous to store (in a `static` variable) the result of the last query.

Searching ordered data (part 1a, array – binary search)

If the data is sorted in increasing (or decreasing) order, searching for a specific value can become a $O(\log n)$ operation. In the case where the information is stored in an array, the search can be performed using an algorithm known as binary search. For following C code presents one of its possible implementations.

```c
int binary_search(T *data,int data_size,T value)
{
  int i_low = 0;
  int i_high = data_size - 1;
  int i_middle;

  while(i_low <= i_high)
  {
    i_middle = (i_low + i_high) / 2;
    if(value == data[i_middle])
      return i_middle; // this may not be the smallest possible index ...
    if(value > data[i_middle])
      i_low = i_middle + 1;
    else
      i_high = i_middle - 1;
  }
  return -1;
}
```

If the data is approximately uniformly distributed, it may be better to select the `i_middle` index by performing a linear interpolation between the points `(i_low,data[i_low])` and `(i_high,data[i_high])`; for an y coordinate of `value` the corresponding x coordinate should be (close to) `i_middle`.

Searching ordered data (part 1b, array – “improved” binary search)

The binary search function presented in the previous slide can be made to return the **first** index `i` for which `data[i] == value` (or -1 if none exists). One possible way of doing this is as follows:

```
int binary_search(int *data,int data_size,int value)
{
  int i_low = -1;         // off by one
  int i_high = data_size; // off by one
  int i_middle;

  while(i_low + 1 != i_high)
  {
    //
    // loop invariants: data[i_high] >= value and data[i_low] < value
    // (by convention, data[-1] = -infinity and data[data_size] = +infinity)
    //
    i_middle = (i_low + i_high) / 2;
    if(data[i_middle] < value)
      i_low = i_middle;
    else
      i_high = i_middle;
  }
  return (i_high >= data_size || data[i_high] != value) ? -1 : i_high;
}
```

[**Homework:** Compare (empirically) the average number of loop iterations performed by the two binary search functions given in this and in the previous slide. Which one is better? Modify the function given in this slide so that it returns the **last** index in case of a match.]

Searching ordered data (part 2, ordered binary tree)

Searching for information in an ordered binary tree can be done in the following way (C code, compare with the unordered case given previously):

```
node *find_recursive(node *root,T value)
{
  if(root == NULL || root->data == value)
    return root;
  return find_recursive((value < root->data) ? root->left : root->right,value);
}
```

It will be a $O(\log n)$ operation if the binary tree is "balanced," but it can be an $O(n)$ operation in the worst possible case if the binary tree is completely unbalanced (with all its left links NULL or all its right links NULL).

A non-recursive implementation is also possible (in this case, it is simpler and faster):

```
node *find_non_recursive(node *root,T value)
{
  while(root != NULL && root->data != value)
    root = (value < root->data) ? root->left : root->right;
  return root;
}
```