16.3 Elementary search methods

In this chapter we have analyzed data structures on which we can efficiently perform operations, such as inserting and removing an element. Still, these structures are not adequate to search for one specific of their elements. We now have a look at how to make this operation efficient, i.e., such that it takes at most $\mathcal{O}(\log N)$ elementary operations for a data set consisting of N items.

16.3.1 *Binary search trees*

In Section 5.2.2 we saw how the binary search algorithm works. Although the binary search on already sorted data is efficient, it is not such an interesting problem because in practice the actual problem is precisely how to get hold of a sorted structure. Usually, the problem consists in maintaining a dynamic data set in a ordered structure to which elements can be added to or removed from. This structure should be sorted enough to allow for an efficient search of an element.

There exist many data structures allowing to store data such that searching an element is an efficient operation. The majority of these structures have a tree-like topology. As we already partially discovered in Chapter 13 and from what we will see here, the cost of managing a tree-like data structure is at most proportional to its maximum depth. This is why the algorithms that have been invented during the last 50 years mainly aim at maintaining a tree as balanced as possible with the least number of operations. To keep our discussion short and concise, we just focus on the simplest structure, the *binary tree*. We refer the reader to more complete textbooks [Cormen *et al.* (1990)] for the treatment of more complicated algorithms, such as *red-black* trees or Fibonacci heaps.

In a binary tree, such as the one in Figure 16.11, each node has (apart from the data item it contains) three pointers: the first one points to the parent node, and the other two to the child nodes, if they exist. By convention, we assign the value `NULL` to the pointers pointing to nodes which are not yet present in the tree (for example, the pointer pointing to the root's parent node and those pointing to the leaves' child nodes). Considering the fact that the node at the root could change when the tree is updated, it is convenient to also keep a variable `root` pointing to the node serving as root.

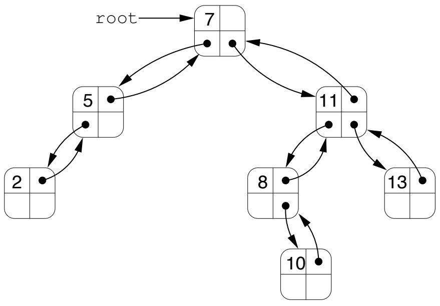

Fig. 16.11 An example of a binary tree. Each node contains the to be stored data item and three pointers defining its family ties. The data have been inserted in the nodes such as to satisfy the fundamental property of a binary search tree.

The structure of the node could be as follows:

```
struct node {
  dataType datum;
  struct node *up, *left, *right;
};
```

where `datum` is a data item stored in the node, `up` points to the parent node, while `left` and `right` to the two child nodes. As usual, we prefer to define a new type `myNode` with the command

```
typedef struct node myNode;
```

to keep the program more concise.

The nodes of a heap structure have a well-defined position in the vector of elements, and therefore also in the physical computer memory. The family ties in a heap are determined entirely by their position in the vector. Instead, in a generic binary tree, the order in which the nodes are written in the memory is irrelevant. Their family ties are made explicit by means of the pointers of each node (analogous to what occurs in linked lists, shown in Figure 16.4). For example, if we want to reserve memory for new tree nodes, this new memory does not necessary have to be contiguous to the already used one. Nevertheless, it is important to realize that once a node has been saved in a certain memory location, it cannot be moved. This is simply because the pointers pointing to that node use that memory location's address. Therefore, we strongly advise against using the command to dynamically reallocate memory (`realloc`) when the data structure is based on the use of pointers.

In order for the search of an element in a binary tree to be an efficient operation, its elements should be sorted such that they satisfy the *fundamental property of binary search trees*: if the data item x is stored in the node `n`, all data items stored in the subtree pointed[2] by `n.left` are less than or equal to x, while all those in the subtree pointed by `n.right` are larger than or equal to x.

As in the case for a heap data structure, also this sorting property allows us to store the same data set in different trees; see for example Figure 16.12. Different trees can be obtained, for example, because the elements could be inserted into or removed from the structure in a different order.

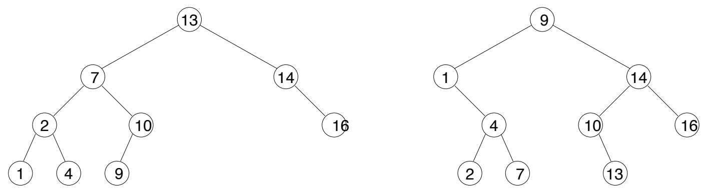

Fig. 16.12 Examples of binary search trees: the same data set can be stored in different ways. Contrary to what occurs in a heap (compare with Figure 16.6), the tree topology identifies the order of the data, whose values increase when read from left to right.

Note, however, that in a binary tree, contrary to what happens in a heap, there is a very strong correlation between the value (or the property) of the data and their position in the tree. Indeed, in both the trees of Figure 16.12, the value of the data increases if the nodes are read from left to right (without considering their depth in the tree). This correlation between the value of the data and their position in the tree is what makes the search for an element in a binary tree efficient.

In the following, we schematically show the main operations on a binary search tree. Keep in mind that the described operations take at most a time which is proportional to the tree's maximum depth $h_{\max}$ (we leave it to the reader to verify this statement), which is typically of the order $\mathcal{O}(\log N)$. So, on average they are efficient operations.

Moreover, none of the functions we are about to describe allocate memory for the tree nodes. Indeed, we assume a node has been "created", i.e., memory space has been allocated, its data fields have been filled and the `NULL` value has been assigned to its pointers, before any of the functions we are about to describe are called.

[2]Slightly abusing notation, we indicate the subtree whose root is pointed by `p` as the subtree pointed by `p`.

16.3.2 *Inserting and searching in a binary search tree*

The *insertion* of a new data item in the tree is a relatively simple operation. Compare the new element with the root of the tree and understand in which subtree it should be placed. Repeat the same operation in the chosen subtree, until you reach an empty subtree, i.e., a space to place the new element. The recursive function in Listing 16.5 performs the insertion of a new node `*new`, whose pointers already have been initialized to `NULL`, in a tree with root `*old`.

To insert a new node `*new` in a tree, we just need to call the function of Listing 16.5 with the pointer to the tree root as a second argument: `insertNode(new, root)`. In the particular case in which we want to insert the first node of the tree, the pointer `root` would be `NULL`. So, we only need to assign it the value `new`, without calling the function `insertNode`.

```
void insertNode(myNode *new, myNode *old) {
  if (new->datum <= old->datum) {
    if (old->left == NULL) {
      old->left = new;
      new->up = old;
    } else {
      insertNode(new, old->left);
    }
  } else {
    if (old->right == NULL) {
      old->right = new;
      new->up = old;
    } else {
      insertNode(new, old->right);
    }
  }
}
```

Listing 16.5 A recursive function to insert a node in a binary tree.

There is a fundamental difference between a heap (that can be represented by a binary tree) and a true binary tree. When representing a heap as a binary tree, the tree topology is fixed and determined only by the heap's size N. Therefore, we immediately know, for example, which elements correspond to the leaves. Instead, a true binary tree has a dynamic topology which is not fully determined by the number N of its nodes. In this case, for example, the nodes without any children cannot immediately be identified. They can be searched by running through the list of nodes or traveling along the tree from the root down. In general, while algorithms operating on a heap can start, in a completely equivalent way, both from the root as from the leaves, those working on trees have a unique access point, namely its root. Alternatively, we could access a tree from a generic node, but this does not contain any information on the position of that node in the tree.

Exercise 3 - Data insertion sequences in a tree

Figure 16.12 contains two different binary trees generated from the same data set, which have been inserted in a different order. Write, for each of the two trees, at least three possible data insertion sequences leading to the filling given in the figure. Note how the insertion sequences compatible with a given binary tree satisfy only partial order relations (which ones?). This should clarify why the trees do not depend only on the data they contain, but also on the way they have been filled.

Exercise 4 - Unbalancing when filling a tree

In case we insert elements with the same value in a binary search tree, the function `insertNode` shows a systematic error because it always places these elements to the left of those already present in the tree. This systematic error could result in very unbalanced trees. How can this problem be solved by slightly changing the function `insertNode`? The solution based on pseudorandom numbers is probably the most obvious one. However, there exists an even easier one not using (pseudo)random variables. Which one?

Hands on 6 - Depth of binary trees

Write a function computing the depth of a node. After filling a tree with N elements of random value by using the function `insertNode`, measure its maximum depth and the average node depth. Compute the expected value of these two depths by averaging over many different noise samples, i.e., the order in which the nodes are added to the tree. Estimate numerically how these two depths grow with increasing N, by a fitting procedure (possibly a linear fit if you choose the right variables!).

Next, consider only the nodes without children, i.e., the tree leaves. As these nodes appear at the end of each branch, they should give a good estimate of the tree depth. Compute numerically the probability distribution of the leaves depth for various values of N (we suggest you to use values in the interval $10^2 \leq N \leq 10^5$ and to compute the averages considering at least a thousand different noise samples). Numerically estimate the average value $\langle h \rangle$ and the variance $\sigma_h^2 \equiv \langle h^2 \rangle - \langle h \rangle^2$ of these distributions. Ho do they grow with N? What can you deduce when comparing them to how the average and maximum value of the node increases? Finally, plot the probability distribution of the variable $z \equiv (h - \langle h \rangle)/\sigma_h$ for various values of N. Observe how the distributions for various N are similar, even though their profile is nontrivial as they are, for example, asymmetric with respect to the origin.

```c
myNode *searchItem(dataType item, myNode *pNode) {
  if (pNode == NULL || item == pNode->datum) {
    return pNode;
  } else if (item < pNode->datum) {
    return searchItem(item, pNode->left);
  } else {
    return searchItem(item, pNode->right);
  }
}
```

Listing 16.6 A recursive function to search a binary tree.

Given a binary search tree filled with data, we now consider a tree search function. To *search* a given element `item` we can use, for example, the recursive function, given in Listing 16.6. The function `searchItem` returns either a pointer to the node containing the searched data item or the value `NULL` if the tree does not contain any such data item. The number of recursive calls is at most equal to the tree's maximum depth in case the function `searchItem` is called with second argument `root`.

Hands on 7 - Non-recursive functions on binary trees

Write non-recursive versions of the functions inserting a node and searching an element in a binary tree. Keep in mind that for simple functions, such as `insertNode` and `searchItem`, it is sometimes preferable to opt for their non-recursive versions because they are faster and not more difficult to program. Still, remember that the performance of an algorithm may depend much on the computer and compiler one is using.

An element which is easy to find in a binary search tree is the *absolute minimum*, because it always coincides with the leftmost element. The function of Listing 16.7 returns the pointer to the node containing the minimum element of the tree (or subtree) whose root is pointed by `pNode`. The maximum element is found with a similar function `maximumInTree`.

```
myNode *minimumInTree(myNode *pNode) {
  while (pNode->left != NULL) {
    pNode = pNode->left;
  }
  return pNode;
}
```

Listing 16.7 A function finding the minimum element in a binary tree.

A last function we need finds the node containing the *successive data item* respect to the one contained in the node given as input to the function. A possible version of this function is given in Listing 16.8.

```
 1 myNode *nextItem(myNode *pNode) {
 2   myNode *pParent = pNode->up;
 3   if (pNode->right != NULL) {
 4     return minimumInTree(pNode->right);
 5   }
 6   while (pParent != NULL && pNode == pParent->right) {
 7     pNode = pParent;
 8     pParent = pNode->up;
 9   }
10   return pParent;
11 }
```

Listing 16.8 A function searching the node containing the immediately successive element.

In the easy case (lines 3-5) the node pointed by `pNode` has a subtree on the right and its successor is simply the minimum element of that subtree. If the subtree on the right does not exist, it is slightly more complicated because we need to search the successive element in a higher node. In this case (lines 6-10) we need to climb the tree with the following rule. For as long as we are climbing towards the left, smaller elements are found and we need to continue to climb. We only stop when we either reach the root (in this case `pParent` equals `NULL`) or if we climb towards the right (in this case `pParent->right` is different from `pNode`). In the first case, we may conclude that the element of which we are looking for the immediately successive one is the largest among those present in the tree. Indeed, in this case the next element does not exist and the function returns the value `NULL`. Instead, in the second case, the next element exists. Namely, it is the one that we find as soon as we start climbing towards to the right. By analogy, it is easy to write the function `previousItem` returning the node containing the immediately preceding element.

16.3.3 *Deleting elements from a binary search tree*

The last operation we need in order to fully be able to manage a binary search tree is the *removal* or *deletion* of an element from the structure. This operation is typically the most complicated one among those needed to manage a data structure, and the binary search tree is no exception to this rule.

Based on the number of children of the node to be removed, we need some different strategies. A simple strategy takes care of the case when it only has at most one child, while a more complicated strategy is needed in case it has two children. In Figure 16.13 we schematically show the effect of removing the gray node in the three possible cases.

In the simplest case, given Figure 16.13(a), the gray node does not have any children and can be simply removed from the tree. The only thing we need to do, is to update the pointer of the parent node, node 4 in the example of Figure 16.13(a), assigning it the value `NULL`[3].

In case the gray node has a single child node, as in Figure 16.13(b), the child node, together with its possible subtree of which it is the root, takes its place when it is deleted. This might seem like a lenghty operation, because an entire subtree is involved. However, if the family ties are managed by means of pointers, we can perform this operation by simply reassigning the pointers of the parent and child nodes of the one we removed (nodes 9 and 4 in Figure 16.13(b)), such that they correctly point to each other. In the subtree with root node 4, the family ties and, as a consequence, the relative pointers remain unchanged. For example, nodes 2 and 7 continue to have node 4 as a parent.

In the third, more complicated case, the node to be removed has two child nodes, Figure 16.13(c), and we cannot simply delete it. Instead, we need to substitute it with another node taking its place without violating

[3]If the tree does not contain any repeated numbers there is a one-to-one correspondence between the tree nodes and the numbers they contain. So, to keep things short, we can indicate the tree nodes by means of the number they contain.