Hierarchical Data Structures (Trees)

Definitions:

A Tree is a hierarchical data structure in which all but one elements have a unique predecessor (parent) but may have many successors (children). The unique parentless element is called the root of the tree. The maximal number of children of an element is called the degree of the tree.

A binary tree is a tree of degree 2.

The elements of a tree are called nodes. Nodes without children are called leaf nodes, the other nodes are called internal nodes.

A path between two nodes is a sequence of adjacent elements starting with the start node and ending with the end node. The length of the unique path between the root and some node is called the depth of the node.

The height of a tree is defined as the maximum of the depth of its nodes.

The path length of a tree is defined as the sum of the depths of all its nodes.

The degree of a node is defined as the number of its children.

A tree is said to be full if all of its internal nodes have the same degree and all of its leaf nodes have the same level.

A tree which is contained in another tree is called subtree of that tree; a tree which contains some other tree is called supertree of that tree.

Tree Traversal for binary trees:

As a binary tree can be interpreted as consisting of the root, the left subtree (having the left child of the root as its root) and the right subtree (having the right child of the root as its root), we can distinguish three forms of traversal:

- inorder traversal (first traverse the left subtree, then visit the root, then traverse the right subtree)
- preorder traversal (first visit the root, then traverse the left subtree, then traverse the right subtree)
- post order traversal (first traverse the left subtree, then traverse the right subtree, then visit the root)

Binary search trees

Definitions:

A binary search tree is a binary tree with the following properties:

- Every node contains a key of an ordinal data type (allowing comparisons)
- For all nodes in the left subtree, the value of the key is smaller than the value of the key of the root.
- For all nodes in the right subtree, the value of the key is greater or equal than the value of the key of the root.

If you perform inorder traversal of a binary search tree, you get the keys in sorted order. Hence, binary search trees can be used for sorting data!

Implementation of the ADT set as binary search tree

Each tree node consists of a key, a link to its left child and a link to its right child. We do not consider a link to the parent node as this link is only needed in some rather special situations.

The binary search tree itself is represented by a reference to its root.

For binary search trees, we will consider three basic operations:

- **Search**
- **Insert**
- **Delete**

Search:

1. If the tree is empty, the result of the search is negative.
2. Else perform steps 2 to 4:
3. If s is equal to the key value of the root, the search is positive.
4. If s is smaller than the key value of the root, then perform the search in the left subtree.
5. If s is greater than the key of the root, then perform the search in the right subtree.

Insert:

1. If the tree is empty, then insert the root node with key s.
2. Else perform steps 2 and 3:
3. If s is equal to the key value of the root, s cannot be inserted.
4. If s is smaller than the key of the root, then insert s into the left subtree.
5. If s is greater than the key of the root, then insert s into the right subtree.

Delete:

1. If the tree is not empty, perform the following steps:
2. If s is smaller than the key value of the root, then perform the deletion in the left subtree.
3. If s is greater than the key value of the root, then perform the deletion in the right subtree.
4. Else perform the following steps:
5. If both subtrees are empty, then delete the root.
6. If the left subtree is empty, then replace the root by the root of the right subtree.
7. If the right subtree is empty, then replace the root by the root of the left subtree.
8. Else perform the following steps:
9. Delete the node containing the minimum key value in the right subtree after having saved the key value (and data).
10. Replace the key value (and data) of the root node by the saved data.

The deletion of the node containing the minimum key value in a binary search tree can be performed as follows:

1. If the left subtree is empty, then perform step 2:
2. Replace the tree by its right subtree and delete the root node.
3. Else perform the operation in the left subtree.

Analysis:

As the time complexity of insertion and deletion of a node in a binary search tree essentially depends on the time complexity of the search for the insert position of the node or the node to delete, we consider only the search.

The worst case occurs if the keys are inserted into the binary search tree in sorted order or reversely sorted order because then the binary search tree degenerates to a linear list where the search has to be performed as a sequential search with time complexity O(n).

The best case occurs if the tree is full because then the height of the tree is ceil(log(n)) and thus the time complexity of the search is O(log(n)).

The average case time complexity of the search operation is 2*ln(2)*log(n) and thus O(log(n)).

Remark:

The analysis is quite similar to the analysis of quicksort!

Dynamically optimized binary search trees (AVL trees)

Binary search trees behave well in the average case but the worst case is really not acceptable. Hence the question how degeneration of a binary search tree can be avoided is of some interest. To investigate this problem, one has to define a measure to decide when something has to be done and of course one has to find some (not too complicated) operation to avoid degeneration.

The first of these methods has been published by Adelson-Velskii and Landis; the trees which are constructed following their method are called AVL trees.

Definition:

For a binary search tree, the balance (number) is defined as the height difference of the left and the right subtree.

A binary search tree is called an AVL tree if for every node, the balance number is –1, 0 or 1.

The AVL property of a binary search tree can be lost when nodes are inserted or deleted so we will have a look at these operations:

Insert:

If the balance number of the tree is –1 (1) and the height of the right (left) subtree is increased, then the tree is balanced, and the height remains unchanged.

If the tree is balanced and the height of the right (left) subtree is increased, then the balance number of the tree is 1 (-1), and the height of the tree is increased.

If the balance number of the tree is –1 (1) and the height of the left (right) subtree is increased, then the height of the tree is increased, the AVL property of the tree is lost, and something has to be done.

Delete:

If the balance number of the tree is –1 (1) and the height of the left (right) subtree is decreased, the the tree is balanced, and the height is reduced.

If the tree is balanced and the left (right) subtree is decreased, the the balance number of the tree is –1 (1), and the height of the tree remains unchanged.

If the balance number of the tree is –1 (1) and the height of the right (left) subtree is decreased, then the height of the tree remains unchanged, the AVL property of the tree is lost, and something has to be done.

Because of symmetry considerations, we need to consider only two cases:

1. The balance number of the tree is –2, the balance number of the left subtree is smaller than or equal to zero. (The latter situation can only occur in the case of a delete!)
2. The balance number of the tree is –2, the balance number of the left subtree is greater than zero.

In the first case, the operation to restore the AVL property is a so called (right) simple rotation:

Let r be the root of the tree, rl the root of the left subtree, LST and RST the left and right subtree (of r) and LSTl and RSTl the left and right subtree of rl, respectively. Then rl becomes root of the tree, r becomes root of the right subtree with left subtree RSTl and right subtree RST, LSTl remains left subtree of r.

After having performed this operation, the following situations are possible:

If the balance number of the left subtree was –1, the tree is balanced after the rotation, and the height of the tree is reduced by one (thus unchanged compared to the last AVL tree before the insert and reduced by 1 compared to the last AVL tree before the delete).

If the left subtree was balanced, the balance number of the tree is 1 after the rotation, and the height of the tree unchanged (thus reduced by 1 compared to the last AVL tree before the delete).

In the second case, the operation of the AVL property is a so called (left right) double rotation:

Let r be the root of the tree, LST and RST the left and right subtree, respectively; let rl the root of LST; let LSTl be the left subtree of rl; let rlr be the root of the right subtree of rl with LSTlr as left subtree and RSTlr as right subtree. Then rlr becomes the root of the tree, rl becomes the root of the left subtree with LSTl as left subtree and LSTlr as right subtree. r becomes root of the right subtree with LSTlr as left subtree and RST as right subtree.

This operation is called RL double rotation because it can be decomposed into a left rotation with pivot rl followed by a right rotation with pivot r.

After having performed this operation, the following situations are possible:

The tree is balanced after the rotation, and the height of the tree is reduced by one (thus unchanged compared to the last AVL tree before the insert and reduced by 1 compared to the last AVL tree before the delete).

If the tree with root rlr was balanced, the left and right subtrees after the rotation are balanced, if the tree with root rlr has negative balance measure, the left subtree has positive balance measure and the right subtree is balanced, if this tree has positive balance measure, then the right subtree has negative balance measure and the left subtree is balanced.

Analysis:

The analysis of AVL trees (worst case time complexity) is usually performed by searching for the AVL trees with minimal number of nodes having height h.

As a binary search tree of height n has the AVL property only if the balance number is –1, 0 or 1, the worst case occurs if the left subtree is the worst AVL tree of height h-1 and the right subtree is the worst AVL tree of height h-2. Hence we get

N(0) = 0,
N(1) = 1,
N(h) = n(h-1)+n(h-2)+1, n>=2.

(The considered trees are called Fibonacci trees, the numbers are called Leonardo numbers.)

One can proof that h(n) <= 1.44 * (log(n+1)+1) and thus the worst case time complexity of search in AVL trees is O(log(n)).

Moreover, one can show that the average case time complexity is about 1.01 * log(n) for n>=1000. This result is nearly identical to the optimum for binary search trees.

Leaf oriented binary search trees

Binary search trees which we investigated until now are node oriented in the sense that every key value occurs in some node, either an internal node or a leaf node.
Sometimes, leaf oriented binary search trees are investigated where every key value occurs in a leaf node of the tree and the internal nodes of the tree are only used for navigation. At the first glance, this seems not very efficient because all keys besides the smallest one are stored twice (once in a leaf node, once in an internal node for navigation). On the other hand, the leaf nodes can be organized as doubly linked sorted list so that direct access is supported as well as sequential access to the data.

Together with the fact that the height of the search tree is reduced because more elements fit into one internal node, this is the reason why (n-ary) search trees for external data structures are usually organized as leaf oriented trees.

For leaf oriented binary search trees, we will consider the standard operations search, insert, delete:

Search:

1. If the tree is empty, then the search is unsuccessful.
2. Else perform the following steps:
3. While the current node is not a leaf node, perform steps 4 and 5:
4. If the search key is smaller than the key value of the current node, set the current node to the root node of the left subtree.
5. Else set the current node to the root node of the right subtree.
6. If the key value of the search key is equal to the key value of the current node, then the search is successful, else the search is unsuccessful.

Insert:

1. If the tree is empty, then perform step 2:
2. Make a new leaf node with key value s and set the tree reference to this node.
3. Else perform the following steps:
4. Search for the (link p to the) leaf node lp where the insertion will take place.
5. Make a new leaf node ln with key value s.
6. If s is smaller than the key value of lp, then perform step 7:
7. Make a new internal node with key value s, left child lp and right child ln.
8. Else perform step 9:
9. Make a new internal node with key value s, left child lp and right child ln.
10. Replace p by a link to the new internal node.

Remark:

Steps 6 and 7 are only necessary if the key value to be inserted is smaller than all key values in the tree. If one can use one dummy node with a key value which is smaller than all possible key values, then these steps become obsolete.

Delete:

1. If the tree is not empty, perform the following steps:
2. If the tree consists of one leaf node, perform step 3:
3. Set the tree reference to undefined (empty tree) and delete the leaf node.
4. Else perform the following steps:
5. Search for the (link lp to the) predecessor p of the the leaf node l to be deleted.
6. If the search is successful, perform the following steps:
7. if l is the right child of p, replace lp by a reference to the left child of p.
8. Else replace lp by a reference to the right child of p.
9. Delete p and l.

B trees and B* trees

If one considers n-ary trees, then the concepts we applied in the case of binary trees do not really work. Hence we will not look at all of these methods but concentrate on the concept of the Btree which has first been published by Bayer and Mc Creight (1970) and is now the method of choice when implementing external tree structures, e.g., in the context of database systems.

(Original) Definition of a B Tree:

An n-ary tree is called a B Tree of degree m if the following statements are true:

1. **All leafs have the same depth.**
2. **The number k of key values in a node is**

   1<=k<=2*m for the root node

   m<=k<=2*m for all other nodes
3. **The number s of successors of a node is**

   s=0 for all leaf nodes

   s=k+1 for all other nodes

These statements must be true during the whole life time of the B tree, hence dynamically managed.

The basic operations (search, insert, delete) can be described as follows:

Search:

1. **Search the node where the key value must reside.**
2. **Search for the key value in the node.**

Insert:

1. **Search for the node where the key must be inserted.**
2. **Perform the insertion of the key into the node.**
3. **If the node is full, perform the necessary overflow handling.**

Delete:

1. **Search for the node where the key value must reside.**
2. **Delete the key from this node.**
3. **If the fill factor is below 50%, perform the necessary underflow handling.**

Overflow handling is usually performed as follows:

1. **If there is a neighbour node with less than 2*m key entries, move some key entries from the current node to this node. The new separating value has to replace the old separating value in the predecessor node. (In practical implementations, this step is often left out.)**
2. **Make a new node. The smallest m key entries are left in the node for which the overflow situation occurred, the largest m key entries are put in the new node, the key entry in the mid is added to the predecessor. If there occurs an overflow in the parent node, an overflow handling for the parent node has to be performed.**

Underflow handling can be performed as follows:

1. **If there is a neighbour node with more than m key entries, then some key entries are moved from the current node to this node. The new separating value has to replace the old separating value in the predecessor node.**
2. **If there is a neighbour node with m key entries, then the key entries of this node and the current node and the separating key entry in the predecessor are put into one node, the other node is deleted, and the keys are organized as in step 2 of the overflow handling during insertion.**

In practical implementations, underflow handling is often left out because usually there are much more insertions than deletions.

Analysis:

First, the time complexity of the search operation is the most interesting result because the time complexity of insert and delete essentially depends on the corresponding result for the search.
The time complexity of the search can be measured as the number of accesses to tree nodes. (In practical applications, the tree nodes are blocks on disk, so the search for the key entry within a tree node can be neglected compared to the access to a tree node.)

To give some impression of the time complexity of B Tree search, consider the case where there are 100 key entries per tree node (and thus about 100 successors for every tree node).
If this assumption is satisfied, one can handle
100 key entries with a B Tree of height 1
10000 key entries with a B Tree of height 2
1000000 key entries with a B Tree of height 3
…

B* Trees:

When Btree implementations are used, e.g., in the context of database systems, usually not the original B Tree structure is implemented, but a slight variation, the leaf oriented version called B* Tree.
The advantages of the B* Tree are the same as the advantages of leaf oriented binary search trees compared to node oriented binary search trees, the possibility of organizing the leaf nodes as doubly linked list so that the sequential access to the key entries is supported as well as the direct access.
Usually, in the nodes of a B* Tree, the internal nodes as well as the leaf nodes, only key values and references to either successors (in the internal nodes) or nodes containing the data associated with the key entries (in the leaf nodes) are stored.
The case of duplicate key values can be solved by organizing overflow lists per leaf node so that no special treatment is needed because the data nodes have to be organized in a similar way. The main problem with this organization is that the association between leaf pages and data pages / overflow pages is lost during a split operation and the reorganization of the associated pages can become quite expensive.

ADT priority queue:

We conclude the chapter about Hierarchical Data structures with the treatment of the ADT priority queue. This ADT can be considered a generalization of stack and queue: The principle of organization is not LIFO or FIFO but best in – first out where some priority is defined to decide which of two elements is "better". (In the case of the stack, the priority is "later insertion", in the case of the queue, it is "earlier insertion".)
When considering which data structure is appropriate for the implementation of the ADT priority queue, one might consider an ordered list (ordered by priority) because then the delete operation is trivial. But the insert operation depending on the search of the insert operation is of linear time complexity so that this choice cannot really be recommended.
The next idea is to use (dynamically optimised) binary search trees but the delete operation depending on the search of the minimum is of logarithmic time complexity as well as the insert operation depending on the search for the insert position, and the case of duplicate priority values has to be considered because most of the methods for dynamic tree optimizations only work when no duplicates are allowed.

Another idea is to use trees, but not search trees but trees which guarantee that the maximal key / priority value is stored in the root node so that the delete operation is trivial as in the case of the ordered list. But the insert can be made more efficient if the tree is organized as a heap:

Definition:

A binary tree is called a heap if the following conditions are satisfied:

1. **For each node, the key value is greater or equal to the key values of the successors.**
2. **The tree is complete.**

Because a heap is complete, there is a "natural" implementation as an array: The root is at position 0, the successors of the node at position N are at positions 2*N+1 and 2*N+2, respectively.

The heap property can be expressed as follows:

- **f [2*N+k] <= f[N], k=1,2,**

where f is the array and the considered indexes are in the index area.
When Heaps are used, e.g., in heap sort, usually this definition of a heap is used in some intermediate steps even if the index area does not start with 0. One has to consider that the two definitions of a heap are identical only if the index area starts with 0 so that the operation must finish with an array starting at index 0.

Heapify algorithm:

The basic operation of heap organization is the insert operation. We consider the case that f is the array representing the heap, f[istart+1], …, f[n-1] satisfy the heap condition and an entry with key value s has to be inserted into the heap so that after the insertion, f[istart], …, f[n-1] satisfy the heap property. The heapify algorithm works as follows:

1. **Start with current index k=istart**
2. **While 2*k+1<n, perform the following steps:**
3. **If 2*k+2 is in the index area of f and f[2*k+2] > f[2*k+1], perform step 4:**
4. **Consider 2*k+2 as reference index (j).**
5. **Else perform step 6:**
6. **Consider 2*k+1 as reference index (j).**
7. **If f[k] < f[j], then perform steps 8 and 9:**
8. **Exchange f[k] and f[j].**
9. **Consider j as current index (k).**

Analysis:

As a heap is a complete tree, the time complexity of the heapify algorithm is log(n).

Heap Sort:

This heapify operation can be used as basic part of a sorting algorithm called heap sort. Heapsort can be compared to selection sort, but using a heap for the maximum search has logarithmic time complexity compared to linear time complexity of the minimum search in the unordered array used in selction sort.

Heap Sort algorithm:

Phase 1 (Building the heap):

1. **Consider f[n/2], …, f[n-1] as heap.**
2. **For k from n/2-1 to 0, perform the heapify algorithm with f[k] as entry to be inserted in the heap and f[k+1],…, f[n-1] as starting heap.**

Phase 2 (Sorting):

1. **For k from n-1 to 1, perform the following steps:**
2. **Exchange f[k] and f[0].**
3. **Perform the heapify algorithm with f[0] as entry to be inserted and f[1], …, f[k-1] as starting heap.**

Analysis:

As the time complexity of the heapify algorithm is logarithmic and this algorithm is applied about n/2 times in phase 1 and n times in phase 2, the time complexity of heap sort is O(n*log(n)) (worst case consideration!).
Hence heap sort can be considered as an alternative to quick sort with its quadratic worst case behaviour and merge sort with its worse space complexity.