Data Structures
Spring 2009

Kishore Kothapalli

Chapter 1

Introduction

1.1 A Gentle Introduction

Trees are an important data structure in Computer Science with applications to optimization, searching, sorting, and compilers. In this chapter, we will first define the tree data structure and then study its applications to compliers via expression trees. We then look at the data structure as an aid to searching. There, we will study resulting efficiency and then study AVL trees which provide an improvement in the efficiency of the search tree.

Definition 1.1.1 *A tree can be defined recursively as follows. A tree is a collection of nodes. An empty collection of nodes is a tree. Otherwise a tree consists of a distinguished node r, called the root, and 0 or more non-empty (sub)trees $T_1, T_2, \cdots, T_k$ each of whose roots are connected by a directed edge from r.*

From the above definition, it holds that in a tree of n nodes, there will be $n - 1$ edges. The fact follows from the observation that each node, except the root, has one edge to its parent and every node has only one parent. Figure 1.1 below shows an example.

In the above figure, A is the root and it is connected to 3 subtrees with roots B, C, and D.

A node with no children is called a leaf node or a pendant node. Nodes with the same parent are called siblings. In the above figure, nodes G, H, I, and K are leaf nodes and nodes C, B, and D, are siblings.

A path from node n_1 to n_k is a sequence of nodes $n_1, n_2, \cdots, n_k$ such that n_i is the parent of n_{i+1}, for $1 \leq i \leq k - 1$. The length of the path is defined to be the number of edges, $k - 1$. A path from a node to itself is said to be a path of length 0. In the above example, a path from C to H is $C -> E -> F -> H$ of length 3.

Notice that in any tree there is exactly one path from the root to any other node.

Given a tree T, let the root node be said to be at a depth of 0. The depth of any other node n_i in T is defined as the length of the path from the root to n_i.

Alternatively, let the depth of the root be set to 0 and the depth of a node is one more than the depth of its parent.

Another notion defined for trees is the height. The height of a leaf node is set to 0. The height of a node is one plus the maximum height of its children. The height of a tree is defined as the height of the root.

In the tree shown in Figure, the height of nodes G, H, I, and K is 0. The depth of C is 1, depth of E is 2. The height of the tree is 5, which is the height of A.

Alike parent-children relationship, we can also define ancestor-descendant relationship as follows. In the path from n_1 to n_2, n_1 is an ancestor of n_2 and n_2 is a descendant of n_1. If $n_1 \neq n_2$, then n_1 is called a *proper* ancestor (descendant) respectively.

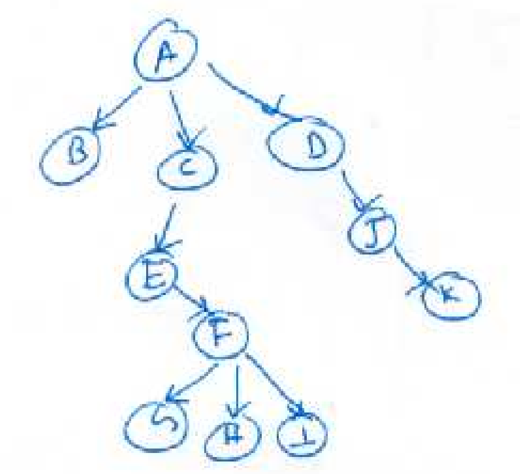

Figure 1.1: A tree with A as the root.

1.2 Implementation of Trees

Briefly, we also mention how to implement the tree data structure. The following node declaration as a structure works.

```
struct node
{
    int data;
    node *children;
}
```

1.3 Binary Trees

A special class of trees is *binary trees*. A binary tree is a tree in which no node can more than two children. The root of the left subtree, it it exists, is called the *left child*, and the root of the right subtree, if it exists, is called the *right child*. Figure 1.2 below shows an example.

In the above figure, the root of the tree is A. Node B is the left child of A and C is the right child. B has D as a left child but no right child. C has F as its right child and has no left child.

The implementation of binary trees is considerably easier compared to general trees as every node has at most two children. Following is an example declaration of a tree node.

```
struct node
{
    int data;
    struct node *left;
    struct node *right;
}
```

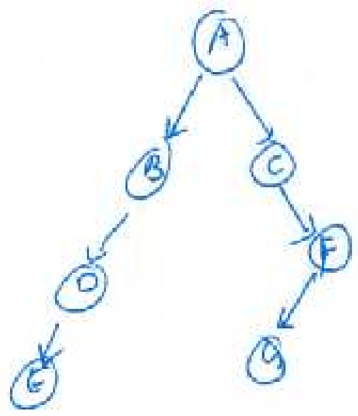

Figure 1.2: A binary tree.

Binary trees have found lots of applications in Computer Science such as searching and compilers. First, we will study an application of trees to compilers via expression trees.

1.4 Tree Traversals

Tree traversal is a technique of listing the nodes in a tree. Let us restrict our attention to binary trees. Consider a recursive definition of a binary tree as having a root, a left (right) sub-binary tree whose root is the left(right) child of the root node. To list all the nodes in the tree, it is required to list the root node, all the nodes in the left subtree, and all the nodes in the right subtree. Denoting by 'D' as the name of the root node, and by 'L' ('R') the node names in the left (right) subtree, we have the following choices.

We could arrange D, L, and R in any order. Thus, there are six choices. However, if we keep the convention that nodes in R are listed only after nodes in L, then we have three choices: DLR, LDR, and RLD. The first choice is called as prefix traversal, the second as infix, and the third as the postfix traversal.

We now illustrate each in detail using the following tree from Figure 1.3 as an example.

The inorder traversal may be completely described by the following procedure written recursively. For the above tree, an inorder traversal results in the sequence $D\ B\ E\ A\ C\ G\ F\ H\ I$.

```
Procedure Inorder($T$)
begin
    if $T \neq$ NULL then
    begin
        Inorder($T->$ left);
        PrintElement($T->$ data);
        Inorder($T->$ right);
    end
End.
```

In preorder traversal, the contents of the root are printed before its subtrees in the left and right order. For the above tree, the preorder traversal sequence is $A\ B\ D\ E\ C\ F\ G\ H\ I$. The procedure for preorder can be written recursively as follows.

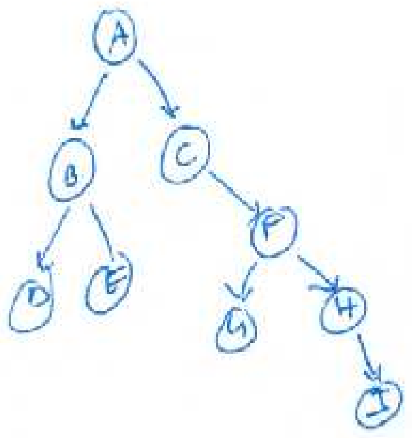

Figure 1.3: A binary tree for illustrating tree traversals.

```
Procedure Preorder($(T)$)
begin
    if $T \neq$ NULL then
    begin
        PrintElement($T->$ data);
        Preorder($T->$ left);
        Prorder($T->$ right);
    end
End.
```

The postorder traversal visits the left and the right subtrees first before visiting the contents of the root. This, for the above tree, the postorder traversal sequence is $D\ E\ B\ G\ I\ H\ F\ C\ A$. The recursive procedure for postorder traversal is as follows.

```
Procedure Postorder($(T)$)
begin
    if $T \neq$ NULL then
    begin
        Postorder($T->$ left);
        Postorder($T->$ right);
        PrintElement($T->$ data);
    end
End.
```

An application of tree traversals is to that of expression evaluation. An expression tree for a expression with only unary or binary operators is a binary tree where the leaf nodes are the operands and the internal nodes are the operators. Figure below shows an example.

To evaluate the expression tree it is meant that we apply the operators to the operands correctly and thereby evaluate the expression corresponding to the tree. The expression corresponding to a tree T can be obtained by an inorder traversal of the tree and adding to indicate precedence. Thus, the expression that

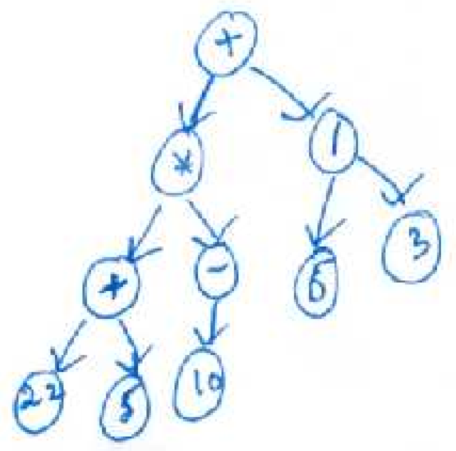

Figure 1.4: An expression tree.

corresponds to the above tree is:

$$((22+5)\times(-10))+(\frac{6}{3})$$

This is what we are used to see when writing expressions. The other two traversals are also used for writing expressions, though rarely. Of the remaining two, the postfix notation has found greater usage especially in Computer Science. In postfix notation, the operands precede the operator. The postfix traversal of the above expression results in $22\ 5\ +\ 10\ -\ 6\ 3\ /\ +$.

While we are normally used to infix notation, the postfix notation introduced by Hamblin turned out to be very useful for evaluating expressions. Some of the reasons are that the paranthesis can be dropped as the operator precedence is taken care of automatically. Evaluation proceeds from left to right, the operands are available before applying the operator.

Notice that the expression tree can be evaluated recursively by first evaluating the left subtree, and then the right subtree followed by applying the operator at the root. The operands are the children. For the above tree, the sequences of trees generated while evaluating the expression tree are:

FIGURE FROM PAGE 7 COMES HERE

After the procedure ends, the remaining tree of 1 node has the result of the expression.

This leads us to the next question. how to create an expression tree. This question assumes importance from the fact that not only is evaluating expressions easier using expression trees, but also code generation for expressions.

We now give an algorithm to construct an expression tree given a postfix expression. The algorithm for converting an infix expression to a postfix expression is covered in an earlier class. The algorithm we study has resemblance to the evaluation of a postfix expression and uses a stack.

Given a postfix expression, we read one symbol at a time from left to right. If the current symbol is an operand, then we push a tree of one node consisting of the operator onto a stack. If the symbol is an operator, then we pop two trees from the stack and create a tree with the root containing the operator and the result of the pop operations as the right and the left subtrees in that order. The resulting tree is again pushed onto the stack.

When we have read all the symbols, the equivalent expression tree is the only element in the stack and a pop operation would give us the tree.

The following example illustrates the algorithm. Let the postfix expression be $a\ b\ +\ f\ -\ c\ d\ \times\ e\ +\ /$. When we first read a and b in that order and create two trees containing node a single node a and b respectively. These trees are then pushed on to the stack in that order. When we encounter the $+$ operator, we pop two trees from the stack, create a tree containing $+$ as the root and the two trees as the right and the left child respectively.

On reading the operand f, we create a tree with f as a single node and push it on to the stack. On reading a $-$ we create the tree with $-$ as the root and the two preivously created trees as its children and push the new tree onto the stack. Similarly, on reading c and d, the stacks has three trees – two new trees corresponding to nodes c and d. On reading a $*$, the stack has two trees – a new tree consisting of $*$ as the root and c and d as its children.

Continuing further, we create the tree with $+$ as the root. On reading the last /, the final tree created is as shown below.

1.5 Binary Search Trees

Annother important application of binary trees is to searching. To illustrate it, consider designing a data structure for primarily three operations: insert, delete, and search. We saw that a hash table can only give average $O(1)$ performance. In this section, we will see how to get worst case $O(\log n)$ performance. We extend the repertoire of operations to standard dictionary operations also such as findMin and findMax. Specifically, our data structure shall support the following operations.

- MakeEmpty()
- Create()
- Insert()
- FindMin()
- FindMax()
- Delete(), and
- Find()

Our data structure shall be a binary tree with the restriction that the value at any node is bigger than the values of all the items in the left subtree and is smaller than the values of all the elements in the right subtree. A binary tree which satisfies the above restriction is called a binary search tree. Figure 1.5 below shows an example of a binary search tree.

The following tree in Figure 1.6 is not a binary search tree as the value at the root is not smaller than the value at its right child.

We now describe how to implement the above operations on a binary search tree. Most of our implementations shall be recursive in nature.

- Find(x) : This operations takes as input an element x and returns a pointer to the node containing data item x in the tree T if x is present in T or returns NULL otherwise. Given the arrangement of data in a binary search tree T, we can implement this routine as follows.

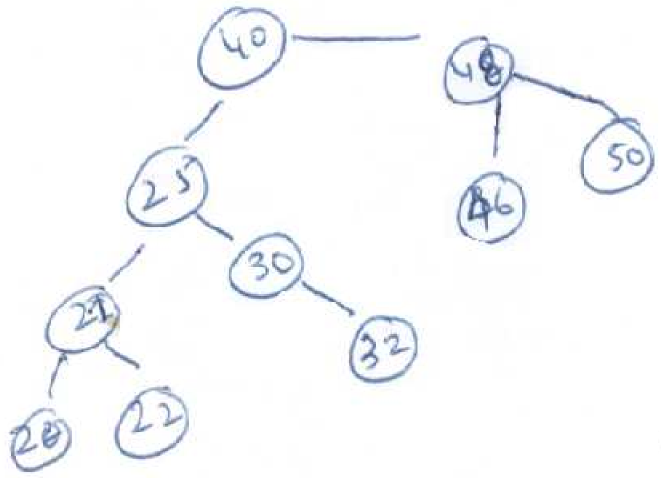

Figure 1.5: A binary search tree.

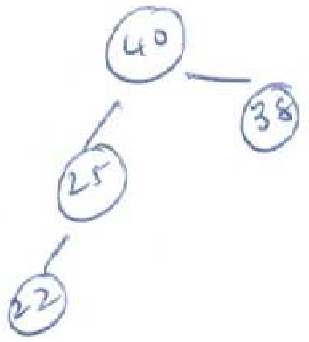

Figure 1.6: A binary tree that is not a binary search tree.

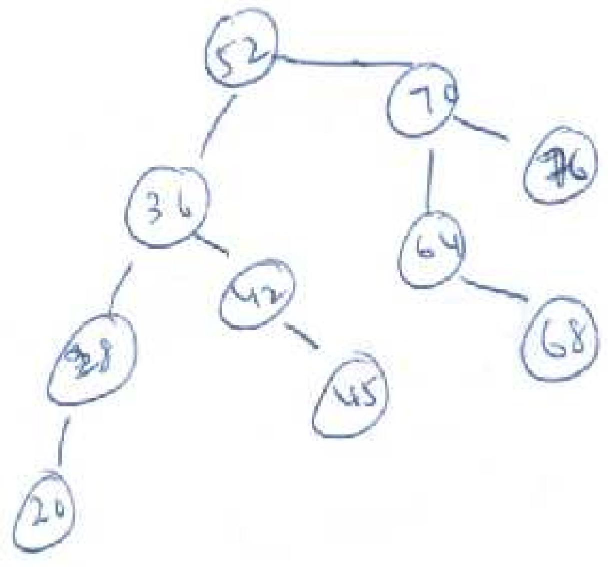

Figure 1.7: A binary search tree to illustrate the Find operation.

Let us compare the value at the root of T with x. There are three possible cases. If the value at T equals x, then the search is successful and we can return a pointer to the root of T. Otherwise, if the value at the root is larger than x, then x can be present only in the left subtree of the root. So we can limit our search to the left subtree, which is also a binary search tree by definition. Similarly, if the value at the root is smaller than x, then we can limit our search to the right subtree of x. If during these recursive calls, we reach a leaf node which is not equal to x, then we can declare the search as unsuccessful.

The following pseudocode illustrates the procedure.

```
Procedure Find($(x, T)$)
begin
    if $T$ is NULL then return NULL;
    else if $T-> data = x$ then return $T$;
    else if $T-> data < x$ then return Find$(x, T-> left)$;
    else if $T-> data > x$ then return Find$(x, T-> right)$;
end End.
```

The following example illustrates the procedure on the tree shown in Figure 1.7. Let x be 76. When calling Find(76), we first compare 52 with 76. Since, $52 < 76$, we call Find$(76, T->$ right$)$ whose root is 70. This time, we again issue a call to the right subtree of 76. Since the root of this tree also has a value of 76, the search is successful.

Notice that in either a successful or unsuccessful search, we traverse only one path from the root. A successful search may stop at some interior node but an unsuccessful search can only stop at a leaf

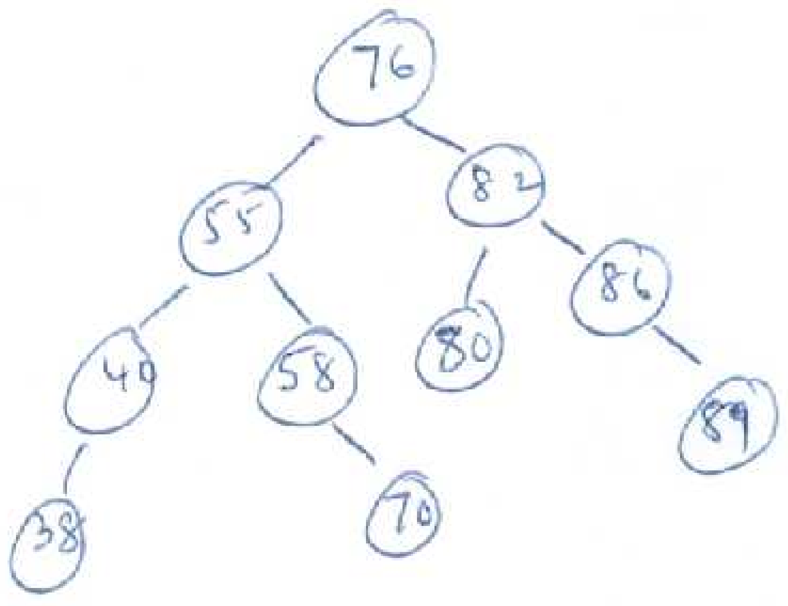

Figure 1.8: A binary search tree to illustrate FindMin and FindMax.

node. This observation allows us to argue that the time taken for the search operation is in the worst case bounded by the depth of the tree T. Even though we implemented the procedure recursively, the same can be written by simply using loops instead of recursion. Also, notice the similarity and differences to binary search.

- FindMin and FindMax : Let us first focus on FindMin. The operation FindMin returns a pointer to the minimum element in the tree.

  Suppose we start at the root of the tree. We know that an element smaller than the root, if it exists, will be in the left subtree. So, we can search recursively in the left subtree. An example illustrates the procedure. Consider the binary search tree shown in Figure 1.8.

  Starting from the root 76, we go to the left subtree with root 55. We then go to the left subtree of 55 with root 40 and similarly to node 38. Once we have exhausted all the left links, we finish the procedure and return 38 as the least element.

  Thus, we traverse a sequence of leftward links till we reach the leftmost leaf node. This operation also takes time equal to the depth of the tree. The following pseudocode implements the above procedure.

```
begin
    if $T$ = NULL return null;
    if $T->$ left = NULL return $T$;
    return FindMin($T->$left);
end
```

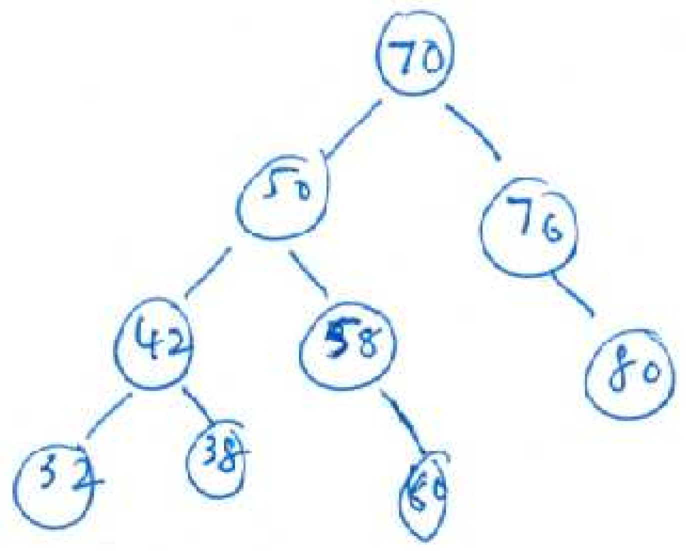

Figure 1.9: A binary search tree.

As with the the Find operation, the procedure can be implemented without using recursion. The FindMax operation is analogous – we need to traverse links in the rightward direction till we reach a leaf node. FindMax also take time proportional to the depth of the tree.

- Insert : To simplify the description, let us assume that duplicates are not part of the input. The procedure to insert a new node into an existing binary tree is similar to that of the Find operation. Starting from the root node, we insert the new new node in the right (left) subtree of the root if the value of the node is larger (smaller) than the value at the root. This comparison is continued till we reach a leaf node ℓ. We insert the new node as a left child of ℓ if $\ell-\!>$data $> x$ and as a right child otherwise.

  If the tree is presently empty, then in this case, we simply create a new binary search tree with the new node as the root.

  The example below illustrates the procedure. Let the element to be inserted be 36, in the tree shown in Figure 1.9.

  To insert node 36, we traverse a sequence of left, left, and left links to reach the leaf node 32. Since $32 < 36$, the node with value 36 is innserted as a right child of node 32. The resulting binary tree is as shown below in Figure 1.10.

  The pseudocode for the above operation is given here.

  Procedure Insert(T, x)

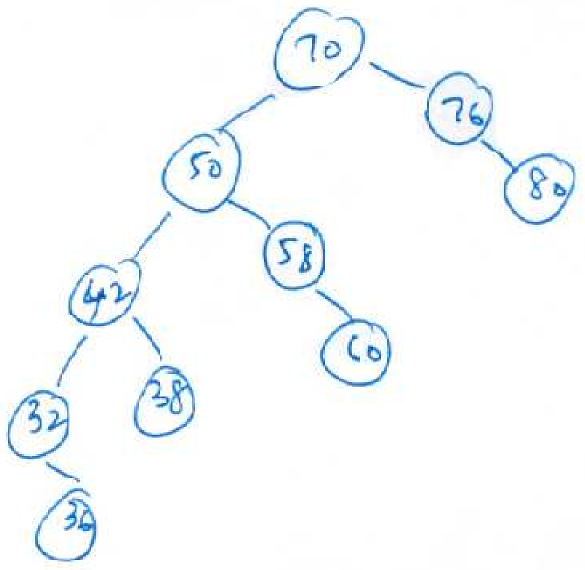

Figure 1.10: The binary search tree after inserting 36 in the tree from Figure 1.9.

```
begin
    $T' = T$;
    if $T'$ =NULL then
        $T'$ = new Node($x$, Null, Null);
    else
        while $T'$ is not a leaf node do
            if $T' -> data < x$ then $T' = T' -> left$;
            else $T' = T' -> right$;
        end-while;
        if $T' -> data < x$ then
            Add $x$ as a right child of $T'$;
        else
            Add $x$ as a left child of $T'$;
        End-if
    End-if
End.
```

It can be noticed that the time taken by the above procedure is also proportional to the depth of the tree.

To handle duplicates, there are several strategies that exist depending on the choice of the implementor. One option is to report an error message indicating that the data structure shall have no duplicates. The second option is to keep track of the number of elements with the same value. The third option of inserting an element in a tree as a duplicate is generally avoided as it can unnecessarily increase the depth of the tree.

- Remove : The hardest operation with respect to binary search trees is the deletion of an existing node. We have several options here. One (easy) option is to perform a *lazy deletion* where the element

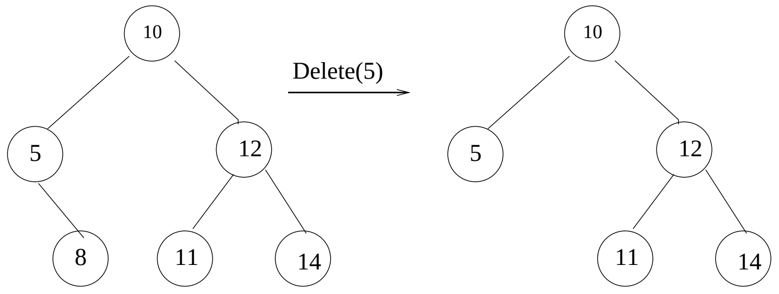

Figure 1.11: A binary search tree along with a deletion of a node with only one child.

to be deleted is not physically deleted from the tree but only marked as deleted. This helps in two settings: one when we expect that the data will reappear shortly, and two when we allow duplicates to be inserted, we can decrement the number of occurrences.

If we have to physically delete the node from the tree, we first find the element using the Find() operation. If the node to be deleted is a leaf node, then we can simply correct the pointer from its parent as NULL and complete the operation. The hardest case is when the node is an internal node. Here again, we have a couple of options.

If hte node has only one child, then the deletion can be completed by adjusting a few links. Specifically, we make the existing node a child of the parent of the node being deleted. The following figure illustrates this case.

If the node being deleted has both a left and a right child, one general approach that is taken is to replace the node being deleted with the smallest node in its right subtree, i.e., the node obtained via FindMin$($node$->$ right$)$. Since such a node is a leaf node, it can be deleted easily and the process completes.

The following example illustrates the process in the tree shown on the leftside of Figure 1.12.

Notice that the replacement does not affect the property of a binary search tree. The node with the smallest value in the right subtree (of a node being deleted) is smaller than any element in the right subtree and is larger than any element in the left subtree.

The pseudocode for the operation is given below. It can be noticed that the time taken by this procedure is also proportional to the depth of the tree.

```
Procedure Delete$(x, T)$
begin
    if $T$ = NULL then return NULL;
    if $T' =$ Find$(x)$;
    if $T'$ has only one child then
        adjust the parent of the remaining child;
    else
        $T'' =$ FindMin$(T'->$ right$)$;
        Remove $T''$ from the tree;
        $T'->$ value $= T''->$ value;
    End-if
End.
```

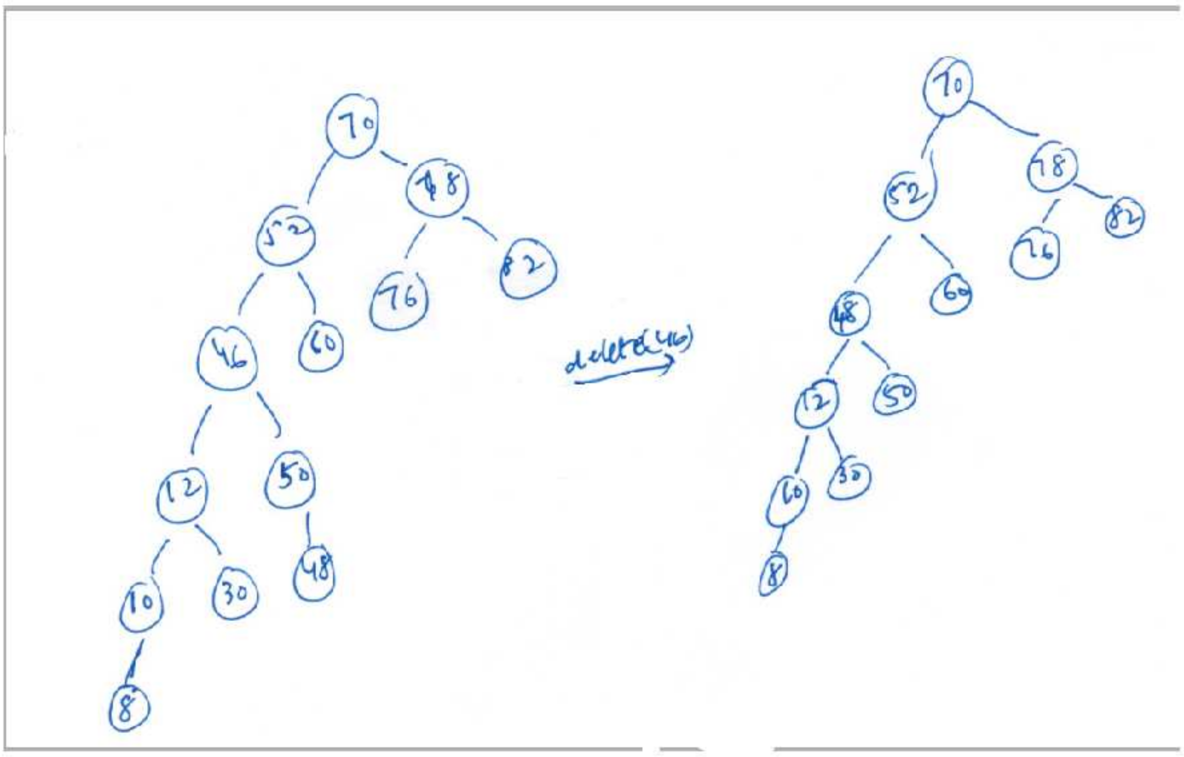

Figure 1.12: A binary search tree along with deletion of a node with both children.

1.5.1 Brief Analysis

So far, while we pointed out that the operations take time proportional to the depth of the tree in the worst case, we did not estimate the depth of the tree. What are some interesting bounds on the depth of a binary search tree? What is the worst case depth and what is the average case depth?

If is intuitive that the average depth of a binary search tree must be $O(\log n)$, for n nodes. It does hold in some cases, which we shall prove now assuming that all (sub)trees over n nodes are equally likelly.

In a tree with n nodes, there is one root node and a left subtree of i nodes and a right subtree of $n-i-1$ nodes. We show that average height of a binary search tree is $O(\log n)$ by arguing that the average internal path length is $O(\log n)$. Internal path length refers to the sum of the depths of all nodes in a tree, i.e., $\sum_{i=1}^{n} d(i)$ where $d(i)$ is the depth of node i.

Let $D(N)$ be the internal path length of some binary search tree of n nodes. We know that $D(1) = 0$. If $D(i)$ is the internal path length of the left subtree and $D(N-i-1)$ is the internal path length of the right subtree. The tree with node $i+1$ as the root node and a left and a right subtree of i and $n-i-1$ nodes respectively has the following internal path length.

$$D(n) = D(i) + D(n-i-1) + n - 1$$

If all subtree sizes are equally likely then $D(i)$ is the average over all subtree sizes. Hence, $D(i) = \frac{1}{n}\sum_{j=0}^{n-1} D(j)$. So is $D(n-i-1)$. Hence, the recurrence relation above simplifies to:

$$D(n) = \frac{2}{n}\left(\sum_{j=0}^{n-1} D(j)\right) + n - 1$$

The above recurrence relation has $D(n) = O(n \log n)$ as its solution. This concludes our argument that under the assumption of all trees being equally likely, the (average) depth of a binary search tree of n nodes is $O(\log n)$.

However, we cannot conclude that all our operations run in $O(\log n)$ time on average. Our delete operation introduces a skew in the tree for the following reason. Our delete operation favours right subtrees over left subtrees for a replacement node and hence can result in right subtrees of shorter depth compared to left subtrees. This skew can be eliminated by choosing the replacement node from the left and the right subtrees unfiormly at random. While intuitively this should solve our problem, the effect of this choice is not known analytically.

Our discussion also suggests that identifying the average scenario is indeed a complex task. If we instead perform lazy deletions, then it does hold that the average depth of the tree is $O(\log n)$. But even then there are some extreme cases which can result in a huge depth. Consider for example, the sequence of insert operations with values in ascending order. This creates a tree where no nodes has a left child, and structurally equivalent to a (sorted) linked list. In this case, all operations take time $O(n)$ in the worst case. To remedy this siutation we need certain balancing techniques, which we will consider in the following section.

1.6 AVL Trees

We saw that binary search trees are a useful aid in seraching provided that the depth of the tree is small, i.e., $O(\log n)$. While this is difficult to ensure in a binary search tree, we will now study a variation to binary search trees that maintains the depth of the tree as $O(\log n)$. This is called AVL tree after its inventors, Adelson–Velskii and Landis.

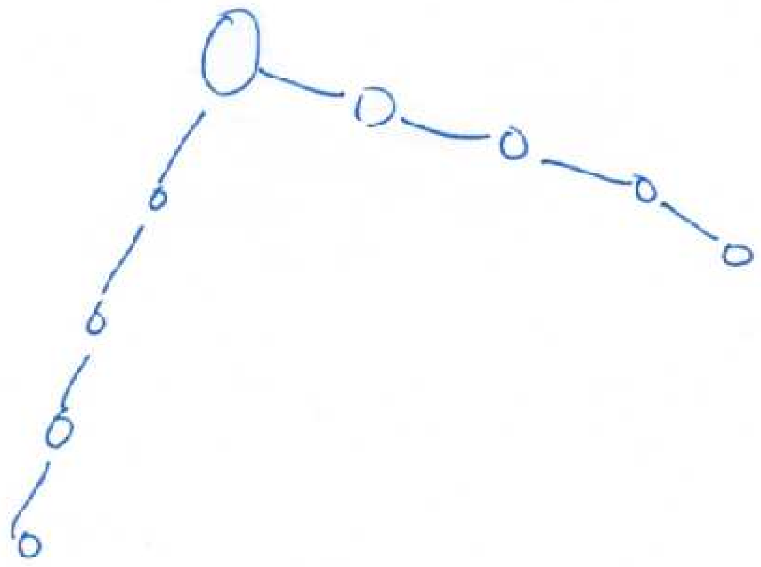

Figure 1.13: A binary search tree.

To ensure that the height of the tree is $O(\log n)$ at all times, firstly notice that insertions and deletions can create problems. One can comp up with an easy solution such as keeping the median of the elements as the root, the median of the first $n/2$ ranked elements as the root of the left subtree, the meain of the last $n/2$ ranked elements as the root of the right subtree, and son on. But this condition is fairly difficult to maintain under insertions and deletions. Each insertion and deletion may alter the entire tree.

What is needed is a simple condition to ensure some sort of balance in the heights of the subtrees and the condition must be easy to maintain during insertions and deletions. One could state the condition as follows.

```
Condition 1: The left and the right subtrees of the root must have the
same height.
```

But it can be noted that the above condition is not enough to guarantee binary search trees with a small depth. Figure 1.13 below shows the reason.

A tightening of Condition 1 stated below might be the next alternative.

```
Condition 2: Every node must have left and right subtrees of the same
height.
```

The above condition is fairly similar to our earlier strategy of keeping the median of the elements in a subtree as the root of the subtree. Thus, this condition, though ensures good balance, is fairly difficult to maintain.

However, a small relaxation to Condition 2 works suprisingly well. The relaxed condition, Condition 3, is stated below.

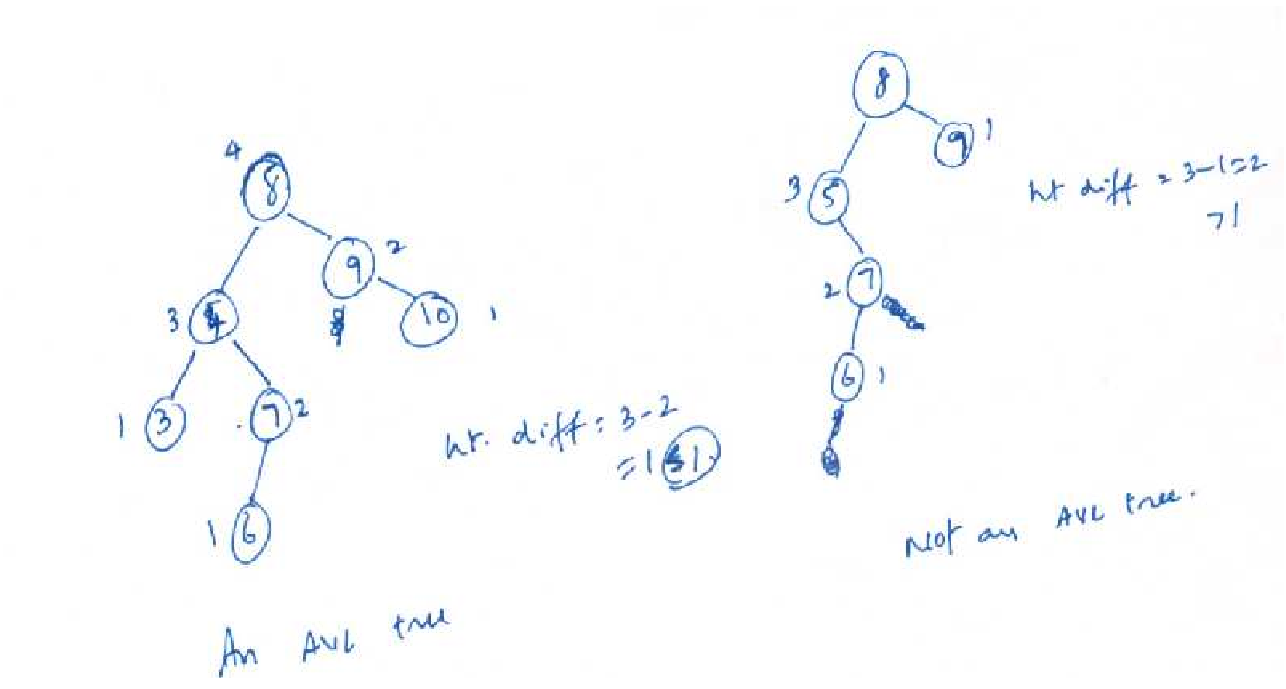

Figure 1.14: A binary search tree.

```
Condition 3: For every node in the tree, the left and the right subtrees
can have heights that differ by at most 1.
```

All throughout, let us delcare the height of an empty tree to be -1.

We define an AVL tree to be a binary search tree with the additional condition called the Balance condition, which is essentially Condition 3 above. Figure 1.14 below shows an exmaple of a binary search tree that is also an AVL tree on the left, and an example of a binary search tree that is not an AVL tree on the right.

To complete our description, we have to specify how to maintain the balance condition after an insertion and deletion. Notice that an empty tree is an AVL tree. For this we introduce an operation, called the *rotation*, that restores the balance condition after an insertion or a deletion.

Notice, for example, that inserting 5 into the AVL tree in the Figure earlier creates a height difference of 2 violating the balance condition. It also holds that after an insertion into an AVL tree, only nodes that are on the path from the inserted node to the root might fail to hold the balance condition. This is because of the fact that only those nodes have their subtrees altered.

At a first glance, it might appear that we may need to restore balance at each node along the new node to the root path. But, we will show that restoring the balance condition at the deepest node where imbalance is found would suffice to satisfy the balance condition throught the entire tree.

Let the node that has to be rebalanced be called t. Then a violation to the balance condition can occur due to these four cases.

- An insertion into the left subtree of the left child of t.
- An insertion into the right subtree of the left child of t.
- An insertion into the left subtree of the right child of t, and

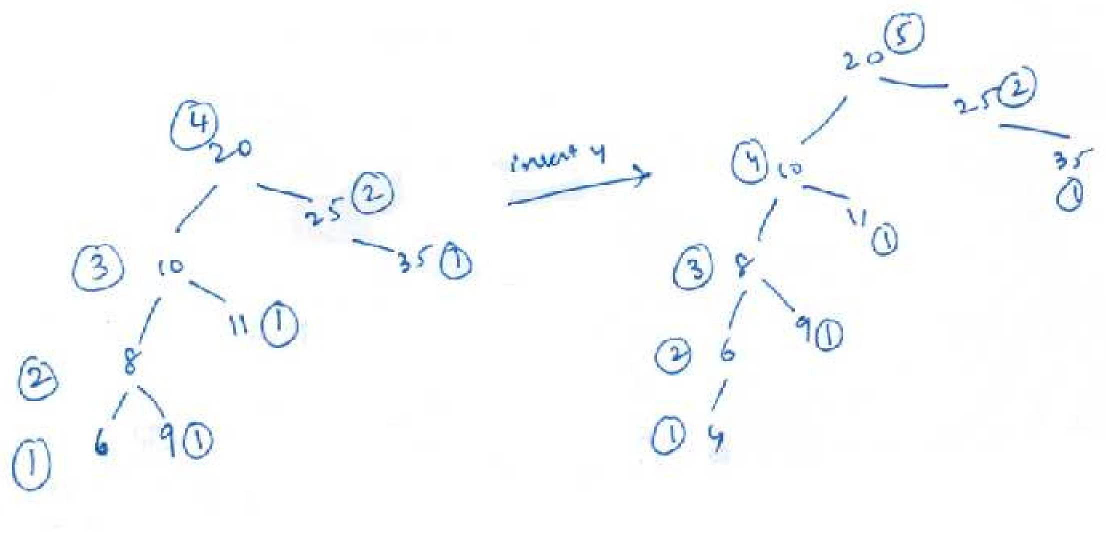

Figure 1.15: A binary search tree before and after inserting 4.

- An insertion into the right subtree of the right child of t.

Of course, we are not considering deletions so far. Of these 4 cases, case (1) and case (4) are symmetric and so are cases 2 and 3. Hence, we are left with only two cases to analyze. In an implementation one has to handle all the four cases, but the treatment is similar.

Cases 1 and 4 turn out to be simple, and are handled in the following manner. Let us look at the tree resulting from the insertion of 4 in the tree shown in Figure 1.15.

The numbers inside the circles are the heights of the various subtrees. It can be observed that the balance condition is violated at node 10. In general, the situation can be captured by the figure shown in Figure 1.16.

Node K_2 has imbalance after an insertion into the subtree X as the height of X increases by 1 and the height difference between subtrees K_1 and Z thus goes up by 2. To restore balance we have to somehow increase the height of subtree Z by 1. To do so, we let k_1 be the root of the subtree currently rooted at k_2. Now k_2 can only be the right child of k_1 as $k_1 > k_2$. This also forces us to keep Z as the right subtree of k_2 and X as the left subtree of k_1. Further, Y can be kept as the left subtree of k_2 as shown.

As we did both reducing the height of X by 1 and increasing the height of Z by 1, we did more than required. Let us now see how the subtree Y should have a height of one plus the height of the tree Z before the rotation, and X should have a height of 1+height of Y before rotation. Otherwise, k_2 would be out of balance even after insertion into X and k_1 would have been the deepest out of balance node in the former.

After the rotation, the height of the subtree rooted at k_2 earlier did not change. So this does not create any cascading sequence of rotations.

The result of single rotation in our example is shown in Figure 1.17 below.

Double Rotation A quick example shown in Figure 1.18 is enough to show that single rotation fails to work in for cases 2 and 3.

Single rotation shifts the imbalance from k_1 to k_2 as the height of the subtree Y does not decrease after the rotation. To solve this problem, we introduce double rotation. It will be helpful to view the subtree as having four subtrees joined to three nodes as there was an insertion to Y implies that it is non-empty prior to the insertion.

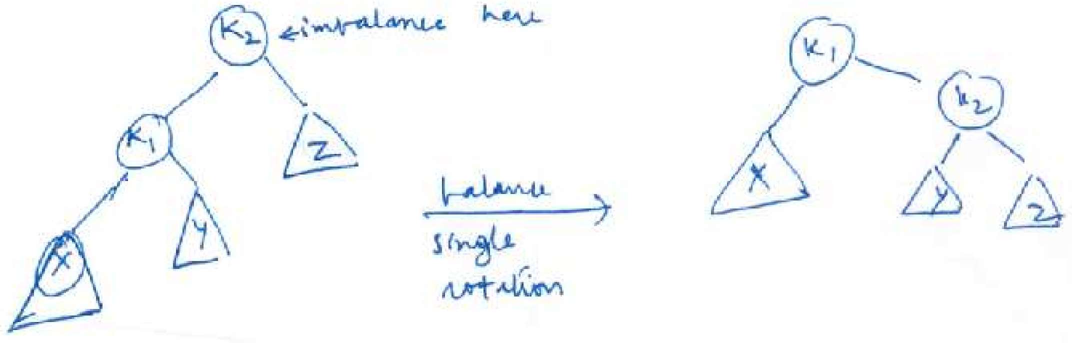

Figure 1.16: Generalization of a single rotation due to an insertion in an AVL tree.

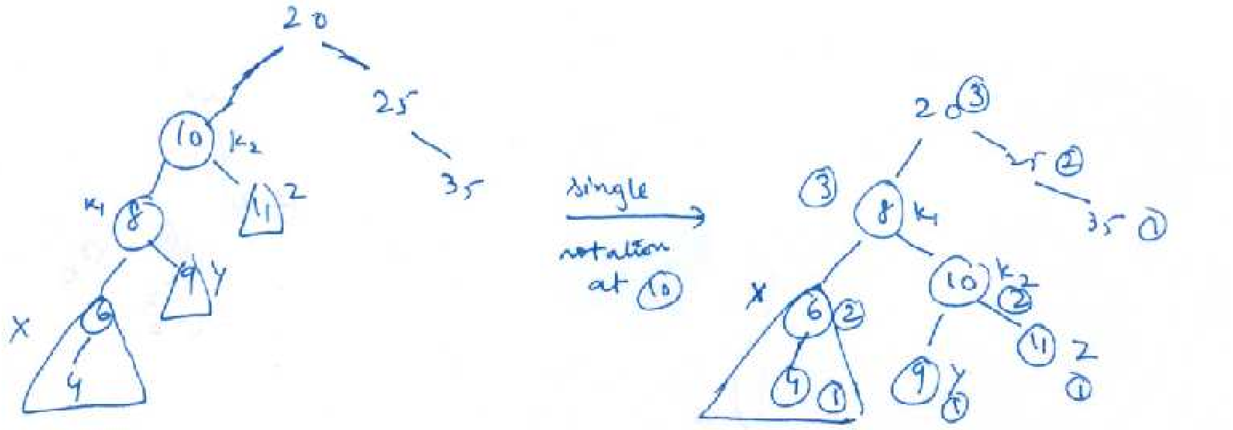

Figure 1.17: An AVL tree illustrating single rotation, resulting due to insertion of 4, and performing a single rotation at node 10.

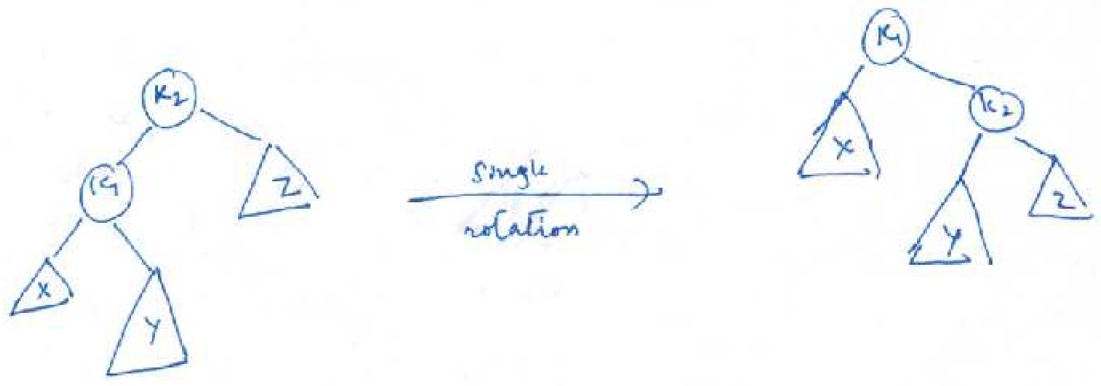

Figure 1.18: Single rotation fails in some cases.

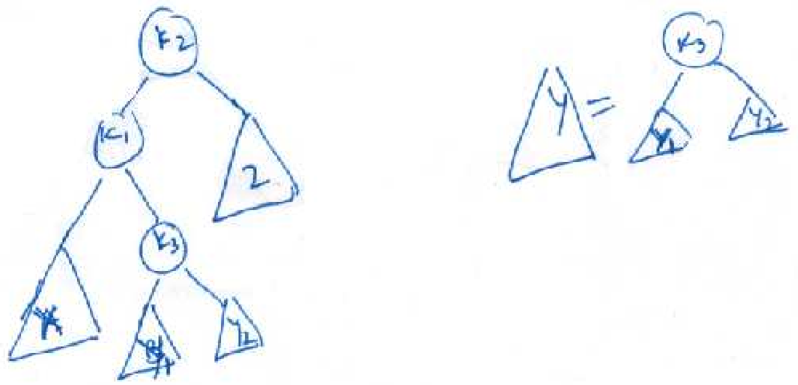

Figure 1.19: A binary search tree.

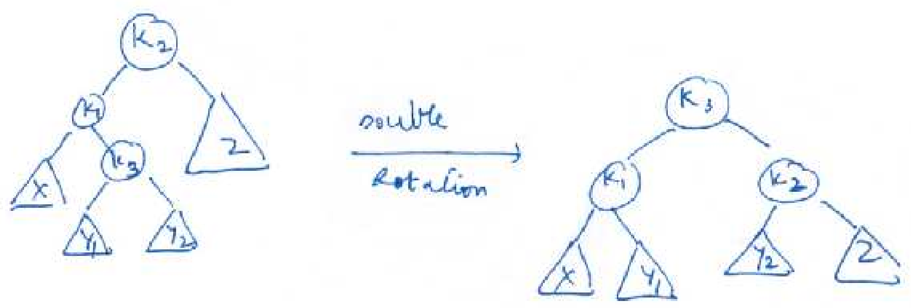

Figure 1.20: An AVL tree along with double rotation.

After the rotation, one of Y_1 and Y_2 are two levels deeper than Z. Though we cannot say which is deeper among Y_1 and Y_2, it turns out that fortunately, it does not matter.

To perform double rotation, we make k_3 as the root of the entire subtree. k_1 will be the root of its left subtree, with X being the left subtree and Y_1 being its right subtree. Similarly, k_2 will be the root of the right subtree of with Z as its right subtree and Y_2 as its left subtree. The resulting tree is shown below.

1.7 Splay Trees

Another variation to height balanced binary search trees is via splay trees. The idea behind splay trees is that while any one operation may be expensive, a sequence of m operations can be performed in a total time of $O(m \log n)$ (assuming $m >> n$). Such data structures are also called as self-adjusting. Compared to AVL trees, such self-adjusting trees may also change the tree during a search operation also. The advantage is that, if it is possible to bring the node that is being searched closer to the root, then latter searches to the same element can finish quickly.

In a splay tree, during every operation, including a search(), the current (serach) tree is modified so that the item searched is made as the root of the tree. During this process, other nodes also change their height. The purpose is to reduce the height of almost all nodes by a factor of 2. This helps in settings where it is likely that the same element may be accessed more often after the first access. Think of caching as an example.

A splay tree is based on the notion of rotations done during an AVL tree insert and delete operation. The details are as follows.[1]

The restructing heuristic is called $splaying$, which moves a specied node to the root of the tree by performing a series of rotations along the (original) path from the node to the root. We have a binary search tree with n nodes and we want to perform m access operations on this tree.To splay a tree at node x, we repeat the following $splaying\ step$ until x is the root of the tree. Let $p(x)$ denote the parent node of x.(The splaying described here is bottom-up).

1. $Case : Zig$ - If $p(x)$ is the root, rotate the edge joining x with $p(x)$.
2. $Case : Zig - Zig$ - If $p(x)$ is not the root, and x and $p(x)$ are both left or both right children,rotate the edge joining $p(x)$ with its grandparent $g(x)$ and then rotate the edge joining x with $p(x)$.
3. $Case : Zig - Zag$ - If $p(x)$ is not the root, and x is left child and $p(x)$ is right child or vice-versa, rotate the edge joining x with $p(x)$ and then rotate the edge joining x with new $p(x)$.

[1]Details mainly from $Self - Adjusting\ Binary\ Search\ Trees$ by D.D.Sleator and R.E.Tarjan

1.7.1 Search

Suppose that T is a binary search tree and it contains the node X that is being searched. For such a successful search, we will now make X as the root of T. This is done iteratively as follows.

Let X be a left child of its parent, Y, which is also a left child of its parent Z. Let A, B, C, D be subtrees so that A and B are the left and right subtrees of X respectively, C is the right subtree of Y and D is the right subtree of Z. To bring X closer to the root, we make X the root of the subtree Z. This also makes Y as its right child, and Z as the right child of Y. While D continues to the right subtree of Z, we now make B the left subtree of Y, and C as the left subtree of Z. This case is also called as a *zig-zig* rotation.

Let X be a right child of its parent, Y, which is a left child of its parent Z. As earlier, let A, B, C, D be trees such that B and C are the left and the right subtrees of X, A is the left subtree of Y, and D is the right subtree of Z. As in the case of a double rotation, we now make X as the root of the subtree Z. This makes Y and Z as its left and right children respectively. Now, A and B are made as the left and the right subtrees of Y, and C and D as the left and right subtrees of Z respectively. This is called as the *zig-zag* case.

The only case remaining is when the parent of X is the root of the tree T. Let Y be the parent of X, and A, B, C be trees so that A and B are the left and the right subtrees of X and C is the right subtree of Y. After a rotation, we make X as the root of the tree and Y as its right child. Subtree A is now the left child of X, and B and C are made the left and the right subtrees of Y.

The zig-zig and the zig-zag cases have their analogous counterparts. In the zig-zig case, X can be the right child of Y and Y is also the right child of Z. We proceed in a similar fashion as earlier. In the zig-zag case, X can be the left child of Y where Y is the right child of its parent, Z. The treatment is similar.

1.7.2 Other Operations

The other operations such as Insert and Delete proceed in a similar fashion. After inserting node X, we splay at node X, so that X is now the root of the tree. After deleting a node X, we splay at the parent of the node that is truly deleted from the tree. If the node deleted has both children, then we replace that node with either its inorder successor or inorder predecessor. Hence, a true deletion happens at a node which is missing at least one child. Let Y be the parent of the node that is truly deleted. The, splay at Y so that after deletion Y is the root of the tree.

1.7.3 How does this help?

Notice that as X is brought to the root of the tree T, also nodes in the path from the root of T to X get closer to the root. Unlike other strategies such as move-to-root, which bring the accessed node X to the root, but make other nodes as deep as X, in a splay tree, most nodes reduce their height after an access.

It can be shown using complicated analysis that *any* sequence of m operations on a splay tree can be completed in time $O((m+n)\log n)$. In other words, while some operation may take more time, the average cost of each operation is $O(\log n)$ once $m \geq n$.

1.8 B-Trees

One of the drawbacks of the approaches seen so far is when the search tree does not fit in the main memory of the computer. In this case, the OS brings pages from disk into the memory as required and writes back from the memory to the disk. Such is the case for instance when using search trees in database applications. In such a setting, it is likely that each record may fall in a different page of the memory. So, even if the search tree is well balanced, a search may require upto $\log n$ disk accesses.

In a computer system, disk accesses are in general much more time consuming than computations. Computations happen at close to the clock frequency, but disk accesses are couple of magnitudes of order slow. So, one can see if this situation can be improved.

Recall that the height of a balanced binary search tree of n nodes is $O(\log_2 n)$. One way to reduce the number of disk accesses is to reduce the height. A possibility in this direction is to increase the branching factor from 2 in the case of a binary tree to $M > 2$. In this case, it can be shown that a balanced M-ary tree on n nodes has a height of $O(\log_M n)$. However, it is not entirely straight-forward to keep an M-ary tree balanced. An M-ary tree can degenerate into a linked list just as a binary tree can. Another danger is that an M-ary tree may degenerate into a binary tree, and we lose the advantage of height reduction. So, more stringent rules are to be maintained so that an M-ary tree does not degenerate into even a binary tree. In the following we describe a method, called a $B+$ tree to arrive at a balanced M-ary tree.

Let M and L be two parameters to be chosen later. An M-ary $B+$ tree on n nodes has the following properties.

- The root of the tree is either a leaf node or has at least 2 children and at most $M - 1$ keys and M pointers.
- Pointer i in any non-leaf node points to the smallest value in the $i + 1$st child of the node.
- Each non-leaf node has at least $\lceil M/2 \rceil$ and at most $M - 1$ children.
- Each leaf node contains at least $\lceil L/2 \rceil$ keys and at most L keys. All leaf nodes are at the same level of the tree. Leaf nodes are arranged in sorted order of keys.
- All data items are stored at the leaf nodes.

An example of a $B+$ tree with $M = L = 5$ is shown in Figure 1.21 below. We now see how to choose M and L. For choosing L, notice that leaf nodes store only records. The basic idea behind the present approach is to place lot of useful infomration in each page. So, if each record is for R Bytes, and a page is of size P Bytes, then we require that each page has $L = P/R$ records. In a similar fashion, a page of P Bytes should contain one non-leaf node. Each non-leaf node has at most $M - 1$ keys and M pointers. If each pointer takes 4 B and each key takes about K bytes, then the total storage for a non-leaf node is $K(M - 1) + 4M$ Bytes. So, we should choose M so that $K(M - 1) + 4M = P$.

We now have to see how to insert, search, and delete from a $B+$ tree. Search is by far the simplest. The root node, if it is not a leaf node, has at least two children. The keys in the root node also indicate the ranage of the values in its children. So, just as in a binary search tree, we locate the right child into which the search proceeds. This process takes us to a leaf node which may contain the key that is being searched for.

In the case of an insert, there are several issues one has to deal with. The easiest case is when the corresponding leaf node has less than L keys. In this situation, the new key is stored in the leaf node and no other changes are required. But what if the leaf node where the new key has to be inserted is already full with L keys? In this case, we split the parent of that leaf node into two nodes. Assume for now that the parent has space for one more key and a link. The leaf node containing L keys plus the new key is split into two leaf nodes each containing $(L + 1)/2$ keys. Both these new leaf nodes have the required minimum

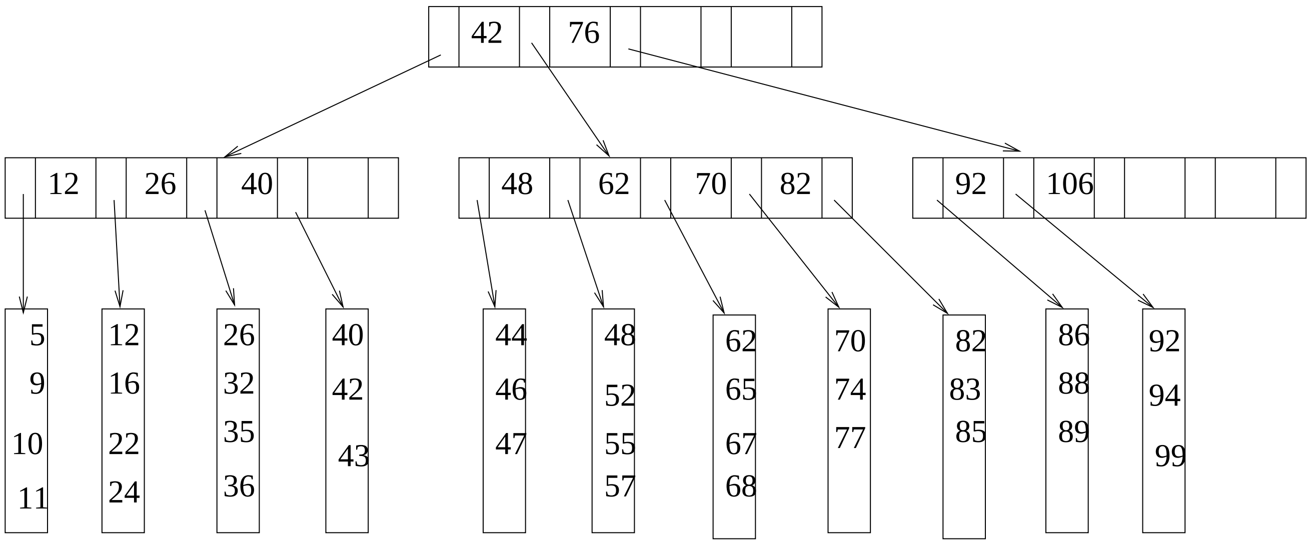

Figure 1.21: A B+ tree with $M = L = 5$.

number of keys. The parent of these leaf nodes will now have another key that points to the smallest entry in the new leaf node. Figure below shows an example.

The process of splitting may result in cascading split operations all the way up to the root. If a non-leaf node is full and cannot accomodate another key, then it is split and its parent is updated appropriately. However, if the root node is full, we use the property that the root node can have two children also and split the root node. Splitting a root node is the only way by which a $B+$ tree gains height.

The process of deletion is similar. As opposed to split, we may have to now merge nodes if they fall below the minimum occupancy.

1.9 Lab and Tutorial

The following shall be discussed in this week's lab and tutorial

- Implementing binary trees and binary search trees.
- Implementing tree traversals.
- Implementing AVL trees.