1. **Data Structures**

1.1 Introduction

A dictionary or priority queue is a set of items with a key attached to each item. Using the key, we can access, insert or delete the item quickly. Examples are a symbol table in a compiler, a database of student academic records, shortest distances maintained by an algorithm dealing with graphs. Thus a dictionary or priority queue can be a part of system program, or directly used for practical purposes, or even can be a part of another algorithm.

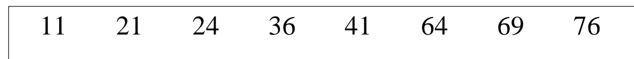
The keys must be from a linearly ordered set.. That is we can answer the question key1 < key2, key1 > key2, or key1 = key2 for any two keys key1 and key2. We often omit the data in each item from an algorithmic point of view and only deals with keys.

A linearly ordered list is not very efficient to maintain a dictionary. An example of linked list structure is given.

A detailed structure of each cell consists of three fields as show below.

If we have n keys, we need O(n) time to access an item holding the key in the worst case. Operations like insert and delete will be easy in the linked structure since we can modify the linking easily.

If we keep the items in a one-dimensional array (or simply an array), it is easy to find a key by binary search, but insert and delete will be difficult because we have to shift the elements to the right or to the left all the way, causing O(n) time. See below.

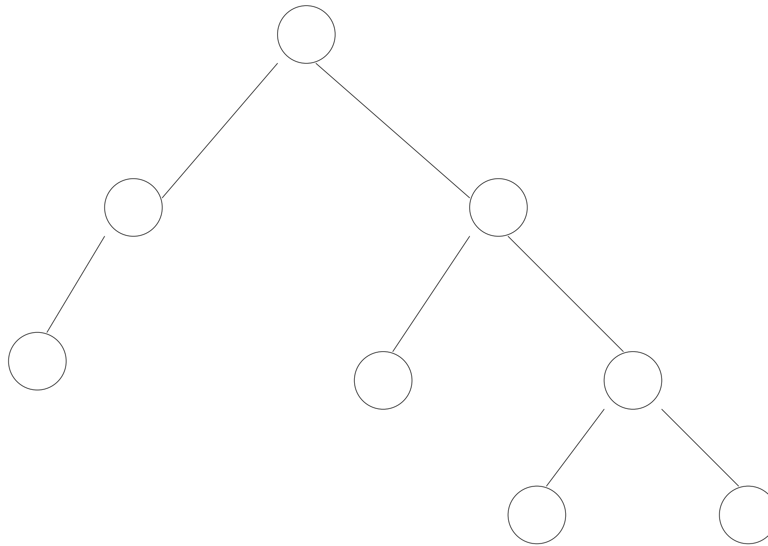


In the linked structure, the keys need not be sorted, whereas in the array version, they must be sorted for finding a key efficiently by binary search. In binary search, we look at the mid point of the array. If the key we are looking for is smaller, we go to the left half, if greater we got to the right half, and if equal we are done. even if the key does not exist, the search will finish in O(log n) time.

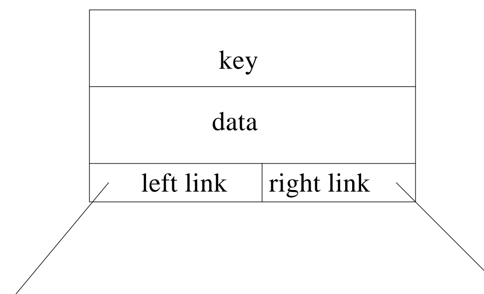
In dictionaries, we implement various operations like “create”, “find”, “insert”, “delete”, “min”, etc. In a priority queue, we do not implement “find”, and try to implement other operations faster.

1.2 Binary Search Tree

Let us organise a dictionary in a binary tree in such a way that for any particular node the keys in the left subtree are smaller than or equal to that of the node and those in the left subtree are greater than or equal to that of the node. An example is given.

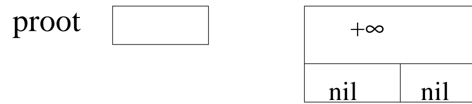


This kind of tree is called a binary search tree. A detailed structure of each node consists of four fields as shown below.

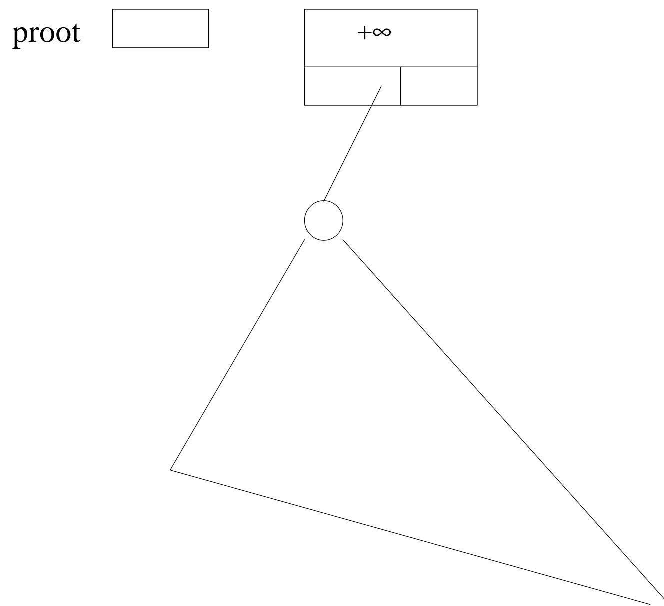


There are many ways we can organise the set of keys {11, 21, 24, 36, 41,64, 69, 76} in a binary search tree. The shape of the tree depends on how the keys are inserted into the tree. We omit the data field in the following.

The operation create(proot) returns the following structure. This stands for an empty tree. We put an a node with the key value of infinity for programming ease.



After we insert several items, we have the following structure.



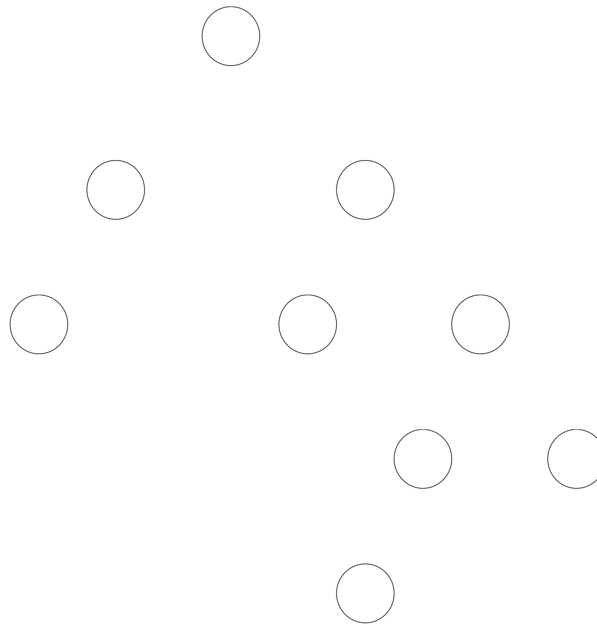
The number of items in the tree is maintained by the variable n. The "find" operation is implemented in the following with key x to be found.

1. **begin** p:=proot; found:=false;
2. **repeat** q:=p;
3. **if** x < p^.key **then** p:=q^.left
4. **else** if x > p^.key **then** p:=q^.right
5. **else** {x = p^.key} found:=true
6. **until** found **or** (p = nil)
7. **end**.
8. {If found is true, key x is found at position pointed to by p
9. If found is false, key x is not found and it should be inserted
10. at the position pointed to by q}

The operation “insert(x)” is implemented with the help of “find” in the following.

1. **begin** find(x); {found should be false}
2. n:=n+1;
3. new(w);
4. w^.key:=x; w^.left:=nil; w^.right:=nil;
5. **if** x < q^.key **then** q^.left:=w else q^.right:=w
6. **end**.

Example. If we insert 45 into the previous tree, we have the following.

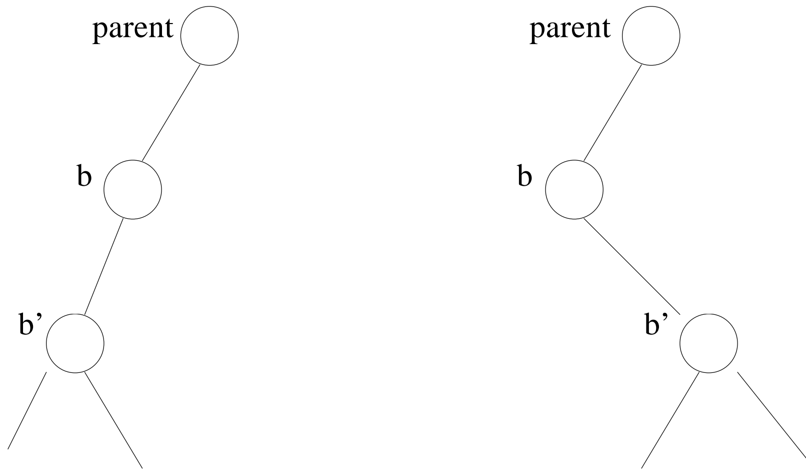


The operation “min” which is to find the minimum key in the tree is implemented in the following.

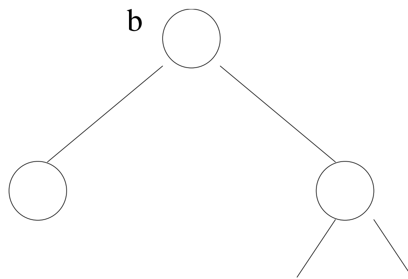
1. **begin** p:=proot^.left;
2. **while** p^.left = nil **do** p:=p^.left;
3. min:=p^.key
4. **end**
5. {The node with the minimum key min is pointed to by p}.

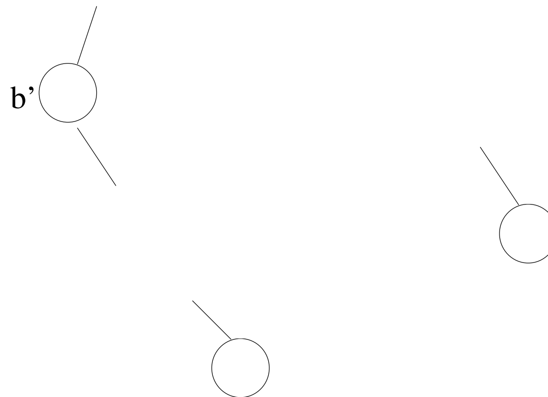
The “delete” operation is implemented with “min”. We delete the node b pointed to by p whose key is x.

If b has only one child, we can just connect the parent and the child. See the following.



If b has two children, we find the minimum b’ in the right subtree of b, and move b’ to b as shown below.

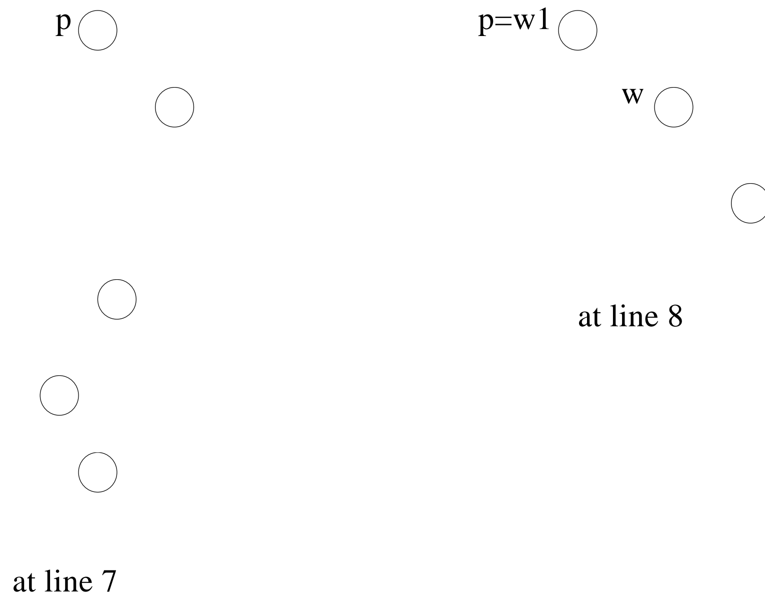




The operation delete is now described in the following.

```
1.  {node  at p whose key is x is to be deleted}
2.  begin  n:=n-1;
3.    if p^.right ≠ nil and p^.left ≠ nil then
4.      begin w1:=p;  w:=p^.right;
5.        while w^.left ≠ nil do
6.          begin w1:=w;  w:=w^.left end;
7.        if w1 ≠ p then w1^.left := w^.right
8.        else w1^.right := w^.right;
9.        p^.key := w^.key
10.       {and possibly p^.data := w^.data}
11.     end
12.   else begin
13.     if p^.left ≠ nil then w:= p^.left
14.     else {p^.right ≠ nil} w:= p^.right;
15.     w1:=proot;
16.     repeat  w2:=w1;
17.       if x < w1^.key then w1:= w1^.left else w1:= w1^.right
18.     until w1 = p;
19.     if w2^.left = p then w2^.left := w else w2^.right := w
20. end.
```

The situations at lines 7 and 8 are depicted below.



at line 7

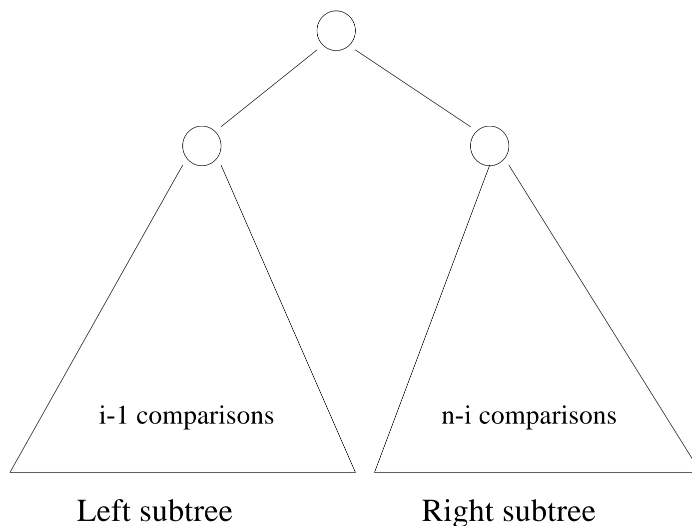
Analysis of n successive insertions in binary search tree

Let us insert n numbers (keys) into a binary search tree in random order. Let T(n) be the number of comparisons needed. Then we have the following recurrence relation.

$$T(0) = 0$$

$$T(n) = n-1 + \quad \Sigma (T(i-1) + T(n-i)).$$

This is seen from the following figure.



Suppose the key at the root is the i-th smallest in the entire tree. The elements in the left subtree will consume i-1+T(i-1) comparisons and those in the right subtree will take n-i+T(n-1) comparisons. This event will take place with probability 1/n.

Solution of T(n)

$$T(n) = n-1 + \quad \Sigma\, T(i) \quad , \text{or} \quad nT(n) = n(n-1) + \Sigma\ T(i)$$

$$T(n-1) = n-2 + \quad \Sigma T(i) \quad , \text{or} \quad (n-1)T(n-1) = (n-1)(n-2) + \Sigma\ T(i)$$

Subtracting the second equation from the first yields

$$nT(n) - (n-1)T(n-1) = n(n-1) - (n-1)(n-2) + 2T(n-1)$$

$$= 2n - 2 + 2T(n-1)$$

That is, $\quad nT(n) = 2(n-1) + (n+1)T(n).$

Dividing both sides by n(n+1), we have

$$T(n) / (n+1) = 2(n-1)/(n(n+1)) + T(n-1)/n$$

i.e., $T(n) / (n+1) = 4/(n+1) - 2/n + T(n-1)$.

Repeating this process yields

$$T(n) / (n+1) = 4/(n+1) -2/n + 4/n - 2/(n-1) + ... + 4/2 - 2/1 +T(0)$$

$$= 4/(n+1) +2(1/n + ... + 1/2 + 1/1) - 4$$

$$= 2 \Sigma \ 1/i - 4/(n+1) - 4$$

That is,

$$T(n) = 2(n+1) \Sigma \ 1/I - 4n$$

$$= 2(n+1) H_n - 4n,$$

where H_n is the n-th harmonic number, which is approximately given by

$$H_n \cong \log_e n - \gamma \quad (n \rightarrow \infty), \ \gamma = 0.5572 \ \text{Euler's constant}$$

That is,

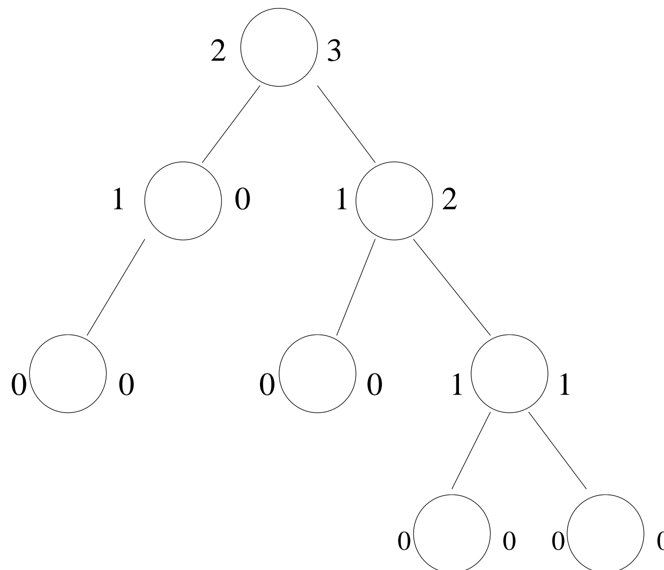
$$T(n) \cong 1.39 \log_2 n -4n \quad (n \rightarrow \infty)$$

1.3 AVL Trees

Roughly speaking, we can perform various operations such as insert, delete, find, etc. on a binary search tree in O(log n) time on average when keys are random. In the worst case, however, it takes $O(n^2)$ time to insert the keys in increasing order. There are many ways to maintain the balance of the tree. One such is an AVL tree, invented by Adel'son-Velskii and Landis in 1962. An AVL tree is a binary search tree satisfying the following condition at each node.

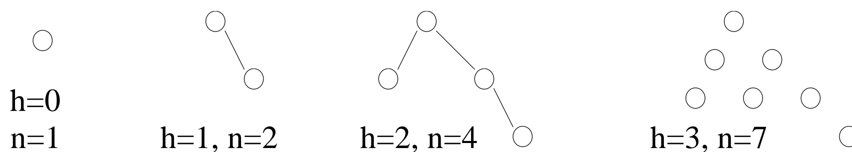
> The length of the longest path to leaves in the left subtree and the length of the longest path to leaves in the right subtree differ by at most one.

Example. The former example is an AVL tree. The longest lengths are attached to each node.

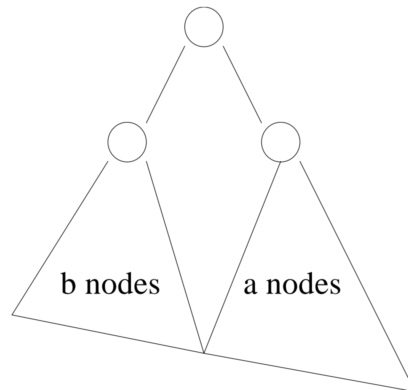


Analysis of AVL trees

Let the longest path from the root to the leaves of an AVL tree T with n nodes be $h_T(n)$, which is called the height of T. Let h(n) be the maximum of $h_T(n)$ among AVL trees T with n nodes. We call this tree the tallest AVL tree, or simply tallest tree. Small examples are given below.



We have the following recurrence for the general situation.



There are n nodes
in this tree

$h(1) = 0$
$h(2) = 1$
$h(a) = h(b) + 1$
$h(n) = h(a) + 1$
$n = a + b + 1$

Let H(h) be the inverse function of h, that is, H(h(n)) = n.. H(h) is the smallest number of nodes of AVL trees of height h. The solution of H(h) is given by

$H(h) = (\alpha^{h+3} - \beta^{h+3}) / \sqrt{5} - 1$
$\alpha = (1 + \sqrt{5}) / 2, \ \beta = (1 - \sqrt{5}) / 2.$

From this we have

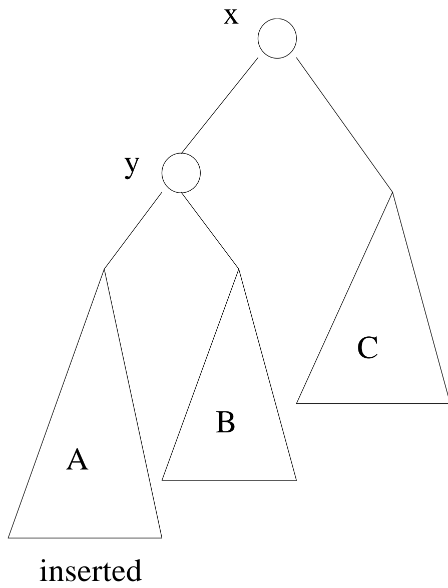
$h \cong \log_{\alpha} n = 1.45 \log_2 n$. That is, the height of AVL trees are not worse than completely balanced trees by 45%.

Rotation of AVL trees

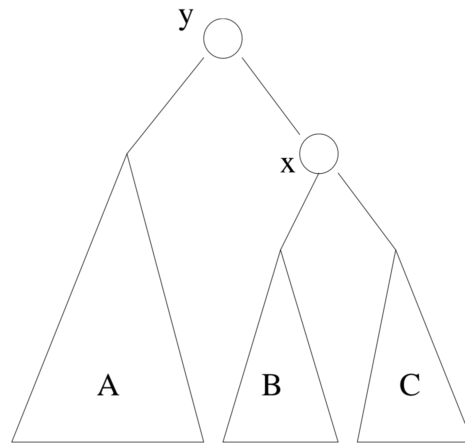
As insert an item into an AVL tree using the insertion procedure for an binary search tree, we may violate the condition of balancing. Let markers L, E, R denote the situations at each node as follows:

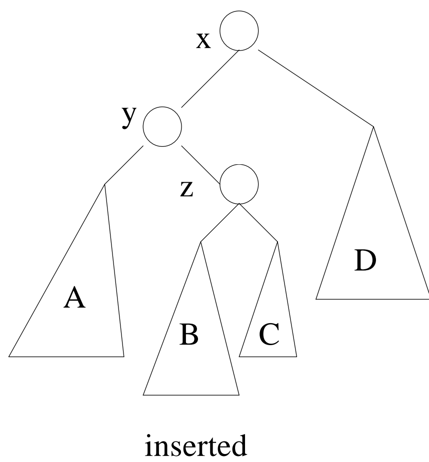
L : left subtree is higher
E : they have equal heights
R : right subtree is higher.

After we insert a new item at a leaf, we trace the path back to the root and rotate the tree if necessary. We have the following two typical situations. Let the marker at x is old and all others are new.

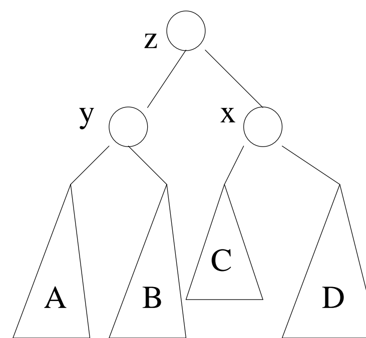


rotate





rotate

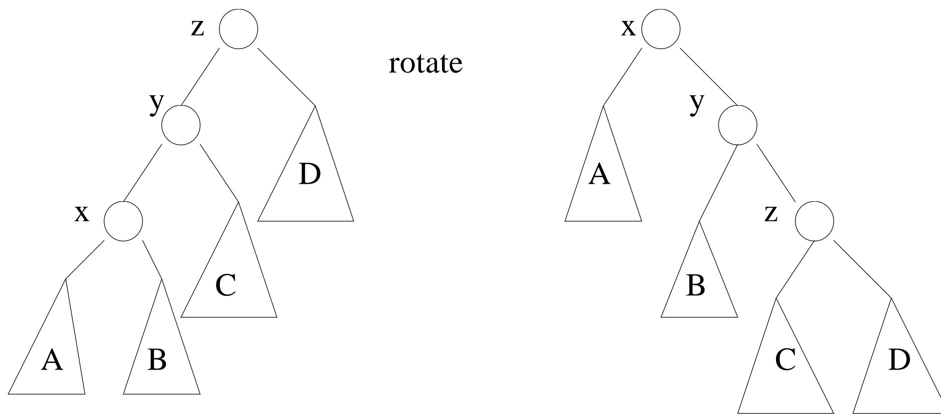


Exercise. Invent a rotation mechanism for deletion.

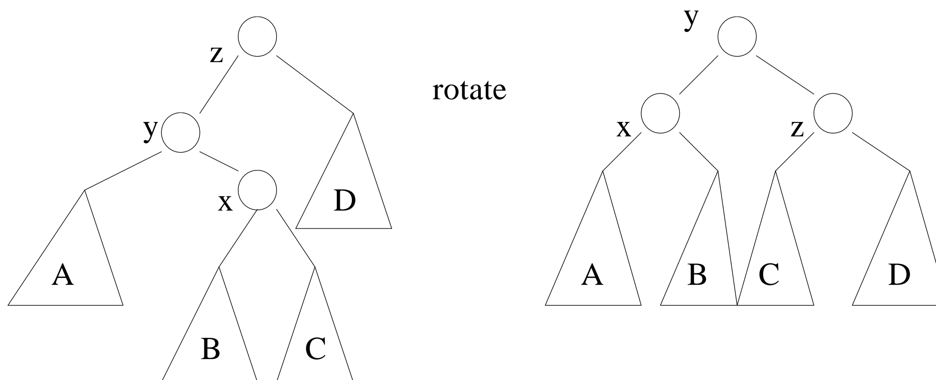
1.4 Splay Trees

A splay tree is a binary search tree on which splay operations are defined. When we access a key x in the tree, we perform the following splay operations. Symmetric cases are omitted.

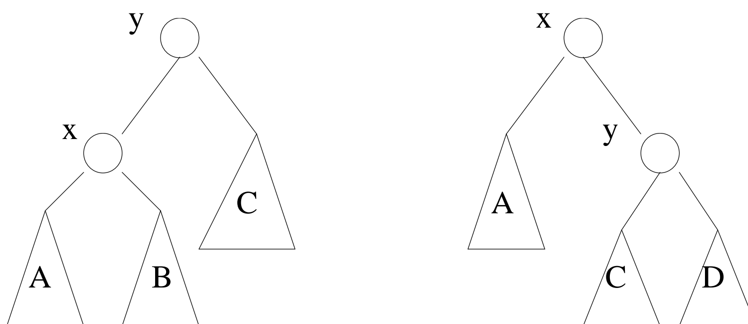
Case 1



Case 2



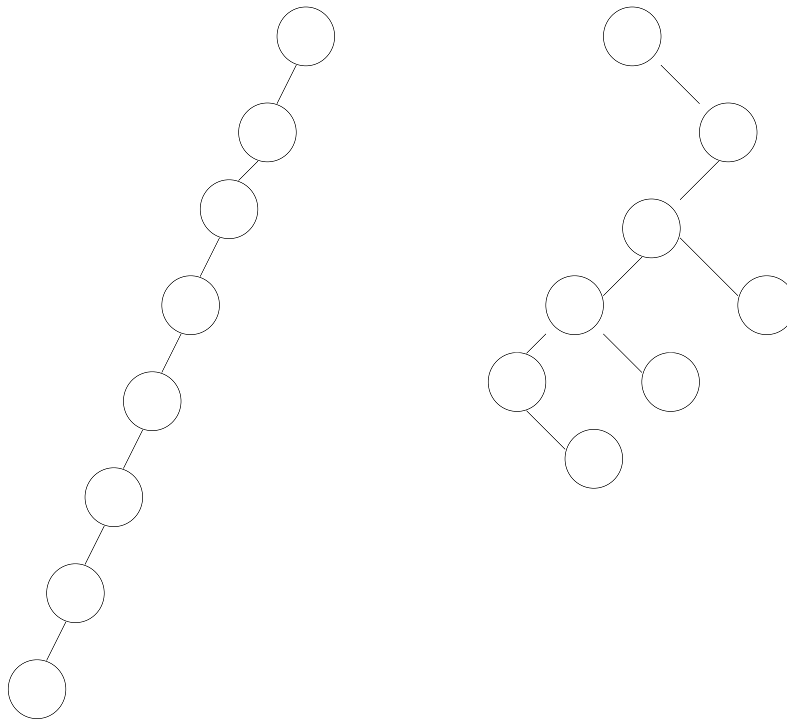
Case 3



We continue case 1 or case 2 towards the root. At the root y, we perform case 3. This whole process is called splaying. The three operations find, insert and delete are described as follows:

find(x) : splay originating at x
insert(x): splay originating at x
delete(x): splay originating at the root of x.

The splay operation contributes to changing the shape of the tree to a more balanced shape. Although each operation takes O(n) time in the worst case, the splay operation will contribute to the saving of the computing time in the future. An example is shown below.



Definition. Amortised time is defined in the following way.

amortised time = actual time - saving

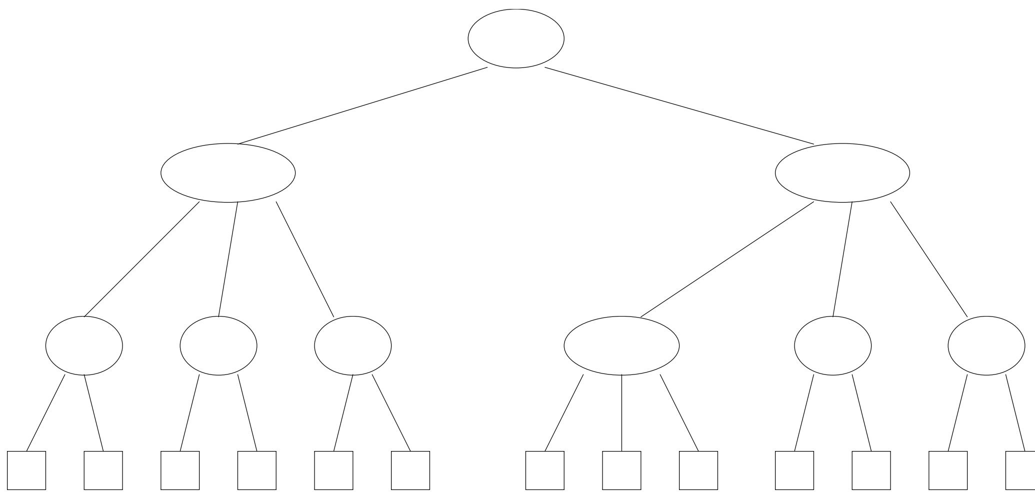
The concept of saving is defined by various means in various algorithms. In splay trees, amortised time is shown to be O(log n). The concept of amortised analysis is important when we perform many operations. It is known that m splay operations take O((m+n)log n) actual time through the amortised analysis..

1.5 B-trees

The B-tree was invented by Bayer and McCreight in 1972. It maintains balance, i.e., the paths from the root to leaves are all equal. There are several variations of B-trees. Here we describe the 2-3 tree in which we maintain items at leaves and internal nodes play the role of guiding the search.

An internal node of a 2-3 tree has 2 or 3 children and maintain 2 or 3 keys.

Example



Internal nodes are round and external nodes (leaves) are square. a node that has i children is called an i-node. An i-node has i keys, key1, key2 and key3 (if any). Keyi is the smallest key of the leaves of the i-th subtree of the i-node.

The operation find(x) is described as follows:

1. Go to the root and make it the current node.
2. Let key1, key2 and key3 (if any) be the keys of the current node.
3. If $x <$ key2, go to the first child.
4. If key2 $\leq x <$ key3 (if any), go to the second child.
5. If key3 $\leq x$, go to the third child (if any).
6. Repeat from 2 until current node is a leaf.
7. If $x =$ key of the leaf, report “found”.
8. If $x <$ key, x is to be inserted to the left.
9. If $x >$ key, x is to be inserted to the right.

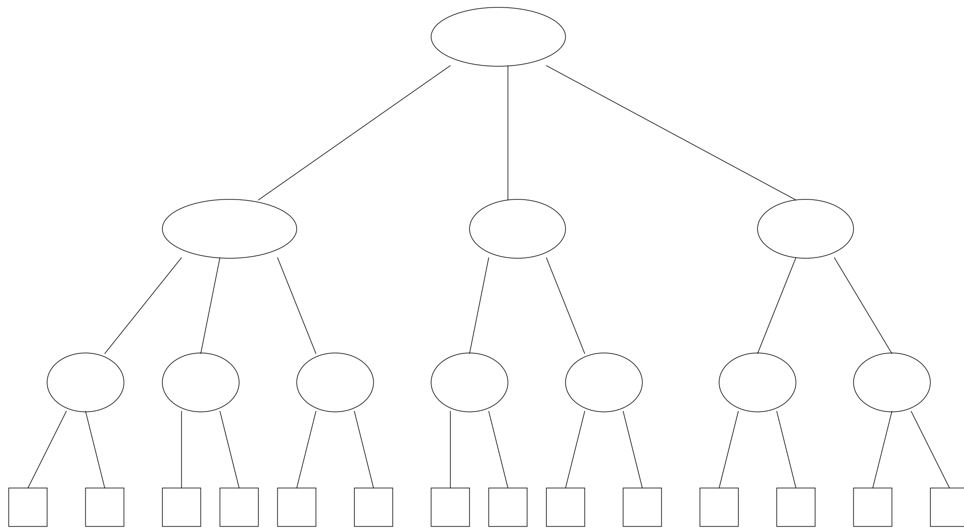
The height of the tree is O(log n) and find(x) takes O(log n) time. Insertion and deletion cause reorganisation of the tree.

If we insert an item at the bottom and the parent had two children originally, there will be no reorganisation, although the keys along the path taken by find(x) will be updated.

Example. If we insert 28, it is inserted to the right of 24 and the parent will have key3 = 28.

If the parent gets four children after insertion, it is split into two and reorganisation starts. If the parent of parent gets four as the results of the splitting, this reorganisation propagates towards the root. The keys over the path will be updated. Key1 is redundant for find(x), but convenient for updating keys held by internal nodes. It is obvious that the whole reorganisation takes O(log n) time.

Example. If we insert 45 into the previous tree, we have the following.



Exercise. Invent a reorganisation mechanism for deletion.