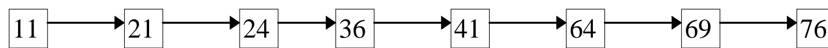
1. **Data Structures**

1.1 Dictionaries

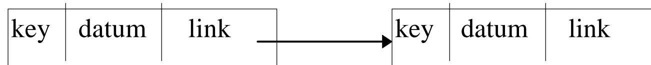
A dictionary is a set of items with a key attached to each item. Using the key, we can access, insert or delete the item quickly. Examples are a symbol table in a compiler, a database of student academic records, shortest distances maintained by an algorithm dealing with graphs. Thus a dictionary can 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.

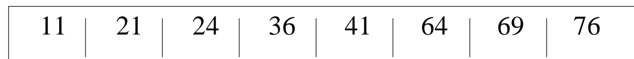


A detailed structure of each cell consists of three fields as show below.



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.

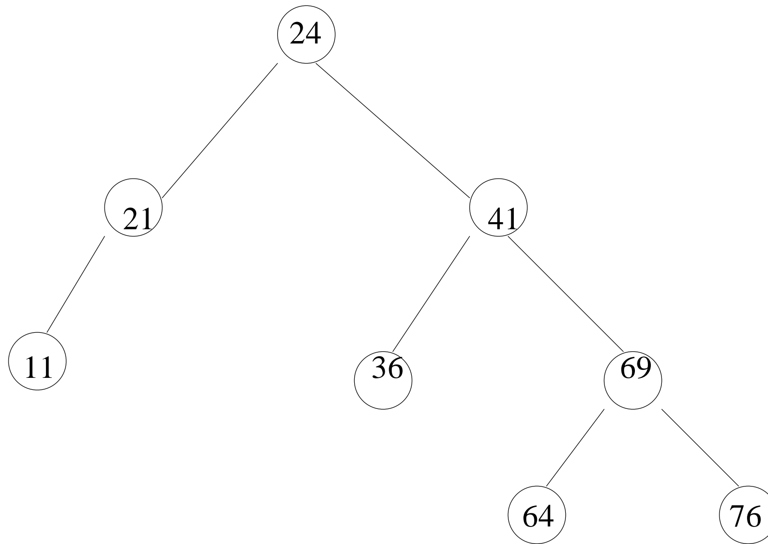


In the linked structure, the keys need not be sorted, whereas in the array version, they must be sorted for finding a key efficiently by binary search. In binary search, we look at the mid point of the array. If the key we are looking for is smaller, we go to the left half, if greater we got to the right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

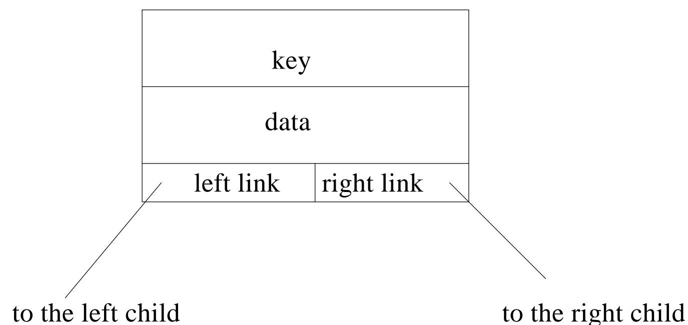
In dictionaries, we implement various operations like “create”, “find”, “insert”, “delete”, “min”, etc.

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.

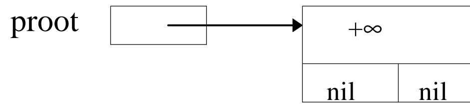


This kind of tree is called a binary search tree. A detailed structure of each node consists of four fields as shown below.

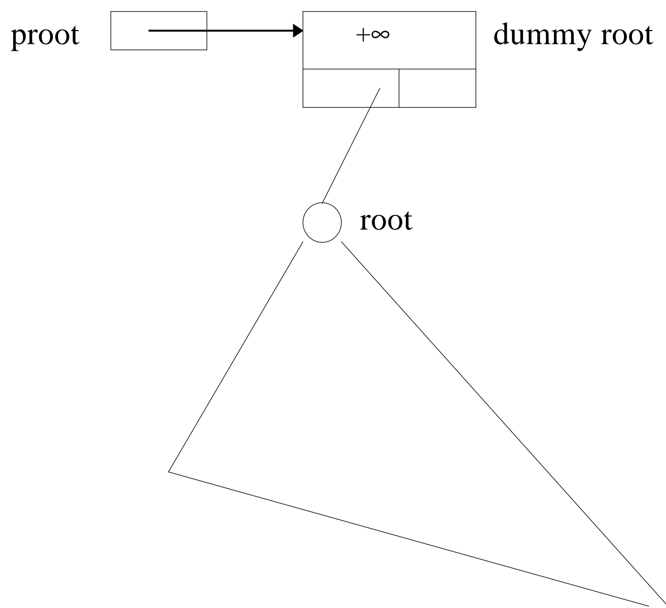


There are many ways we can organise the set of keys {11, 21, 24, 36, 41,64, 69, 76} in a binary search tree. The shape of the tree depends on how the keys are inserted into the tree. We omit the data field in the following.

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.



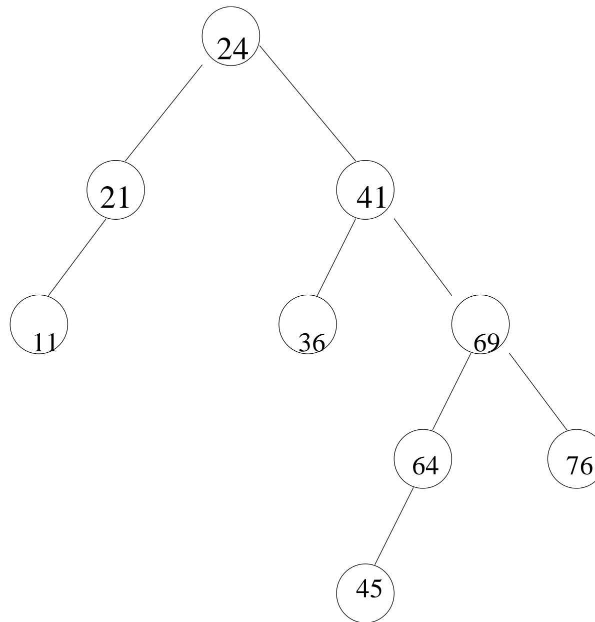
The number of items in the tree is maintained by the variable n. The “find” operation is implemented in the following with key x to be found.

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 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.

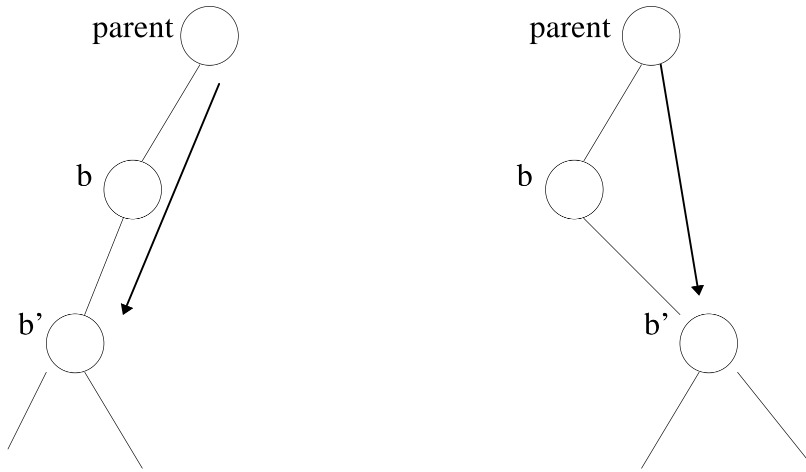


The operation "min" which is to find the minimum key in the tree is implemented in the following.

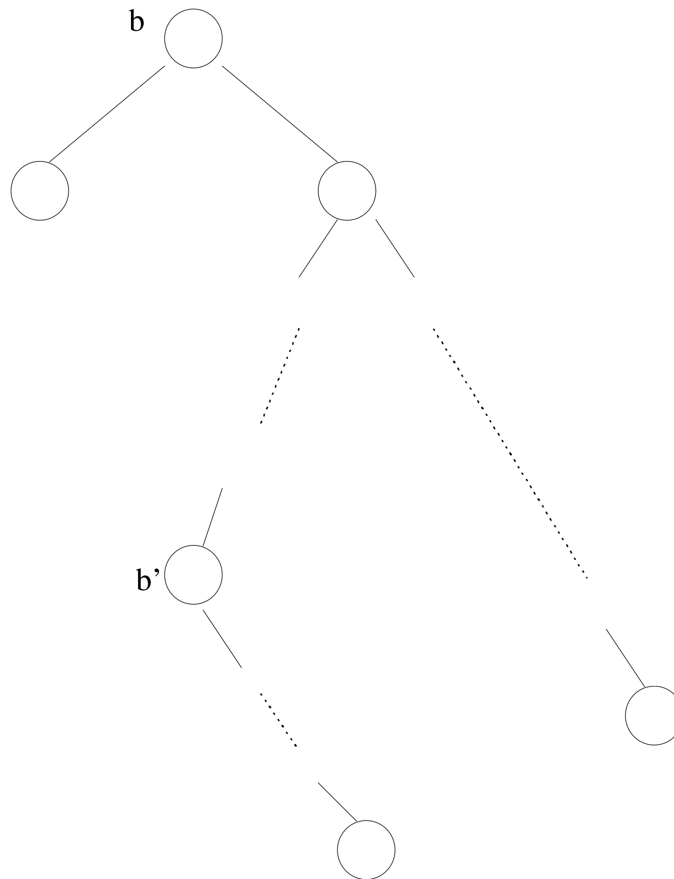
1. **begin** p:=proot^.left;
2. **while** p^.left = nil **do** p:=p^.left;
3. min:=p^.key
4. **end**
5. {The node with the minimum key min is pointed to by p}.

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.



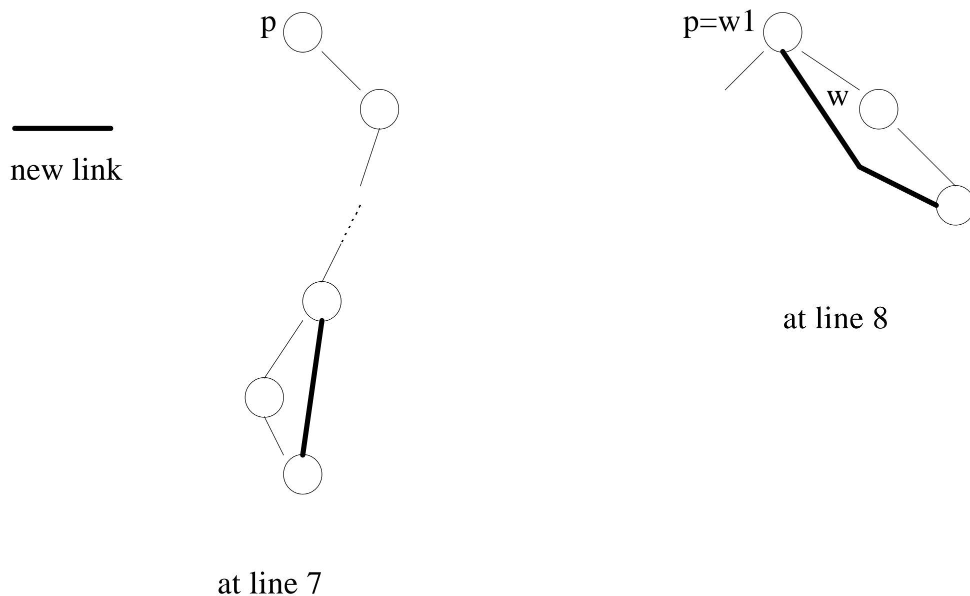
If b has two children, we find the minimum b’ in the right subtree of b, and move b’ to b as shown below.



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.



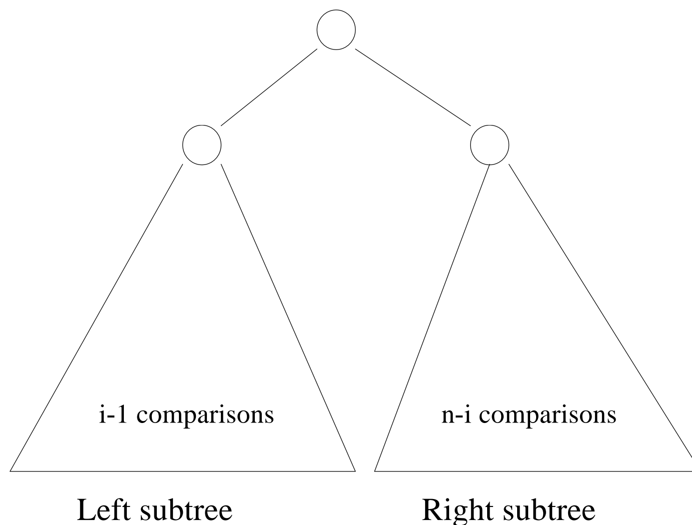
Analysis of n successive insertions in binary search tree

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.

$T(1) = 0$

$T(n) = n-1 + (1/n) \Sigma_{[i=1 \text{ to } n]} (T(i-1) + T(n-i)).$

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)

$T(n) = n-1 + (2/n)\Sigma_{[i=0 \text{ to } n-1]} T(i)$, or $nT(n) = n(n-1) + 2\Sigma_{[i=0 \text{ to } n-1]} T(i)$

$T(n-1) = n-2+(2/(n-1))\Sigma_{[i=0 \text{ to } n-2]} T(i)$, or $(n-1)T(n-1) = (n-1)(n-2)+2\Sigma_{[i=0 \text{ to } n-2]} T(i)$

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,

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

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

Repeating this process yields

$$T(n) / (n+1) = 4/(n+1) -2/n + 4/n - 2/(n-1) + ... + 4/3 - 2/2 +T(1)$$

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

$$= 2\Sigma_{[i=1 \text{ to } n]} 1/i + 4/(n+1) - 4$$

That is,

$$T(n) = 2(n+1) \Sigma_{[i=1 \text{ to } n]} 1/i - 4n$$

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

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

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

That is,

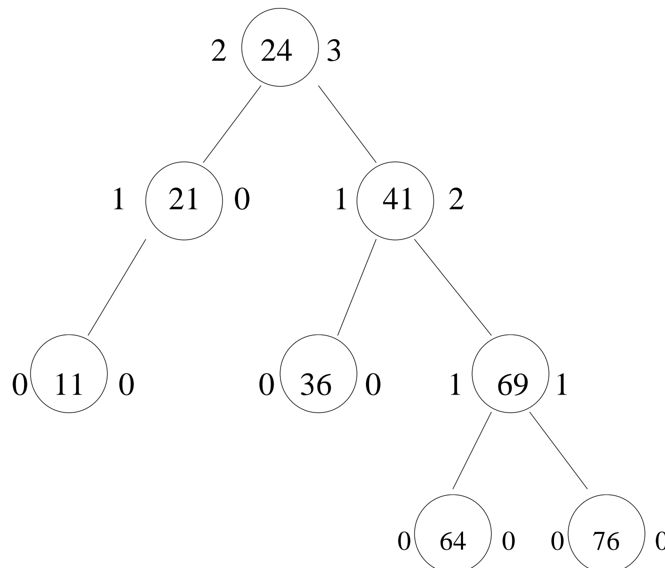
$$T(n) \cong 1.39 \log_2 n -4n \quad (n \to \infty)$$

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.

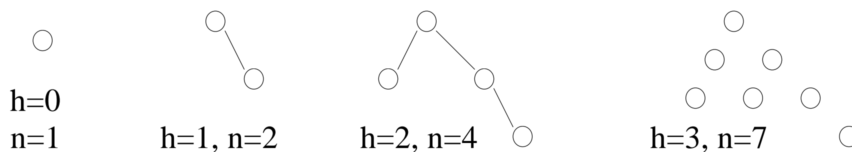
> The length of the longest path to leaves in the left subtree and the length of the longest path to leaves in the right subtree differ by at most one.

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

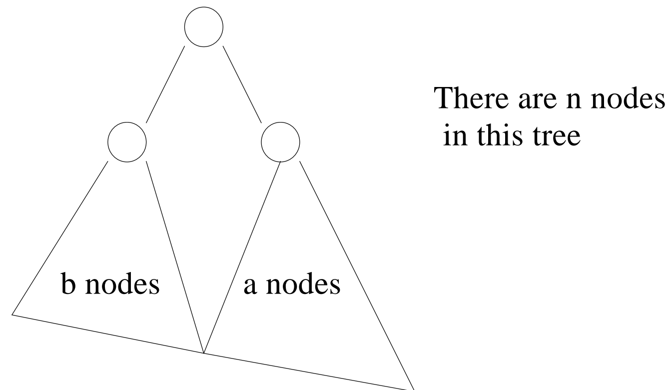


Analysis of AVL trees

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.



$h(1) = 0$ $\quad$ $H(0) = 1$
$h(2) = 1$ $\quad$ $H(1) = 2$
$h(a) = h(b) + 1$
$h(n) = h(a) + 1$
$n = a + b + 1$ $\quad$ $H(h) = H(h-1) + H(h-2) + 1$

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

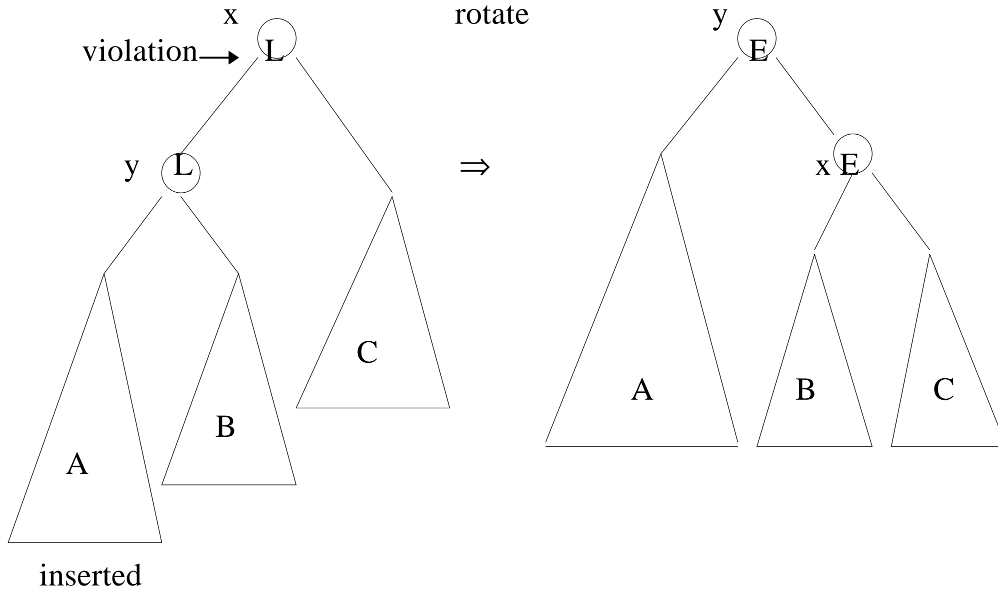
$h \cong \log_{\alpha} n = 1.45 \log_2 n$. That is, the height of AVL trees are not worse than completely balanced trees by 45%.

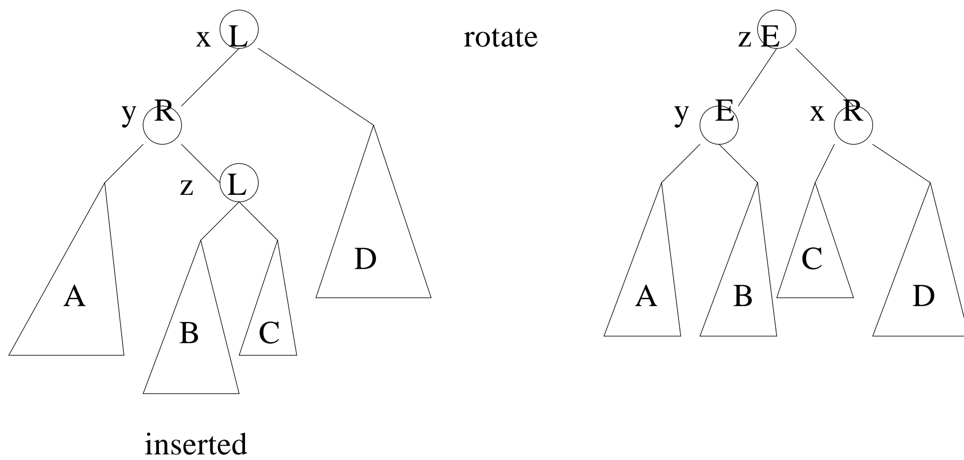
Rotation of AVL trees

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.

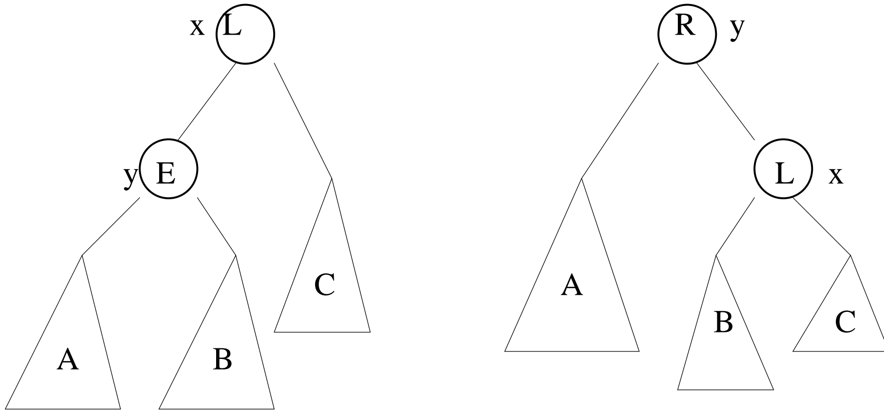




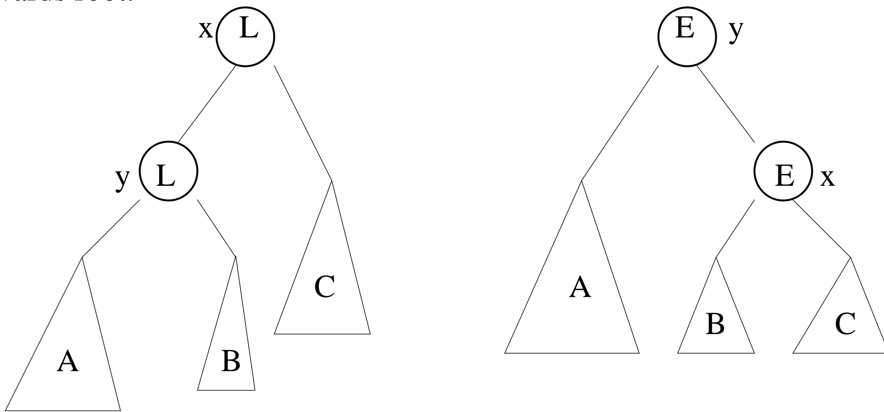
Exercise. Invent a rotation mechanism for deletion.

Deletion in an AVL Tree

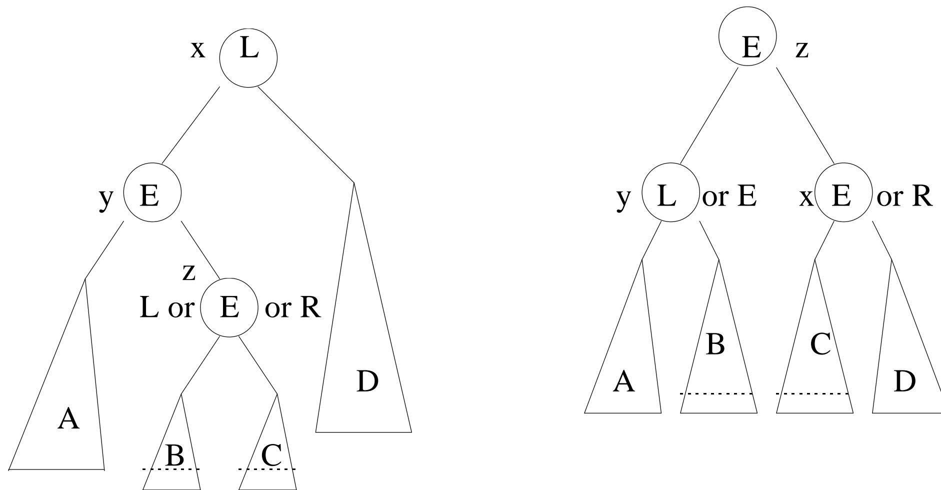
Case 1. Deletion occurred in C. After rotation, we can stop.



Case 2. Deletion occurred in C. We lost the height by one, and need to keep adjustment towards root.



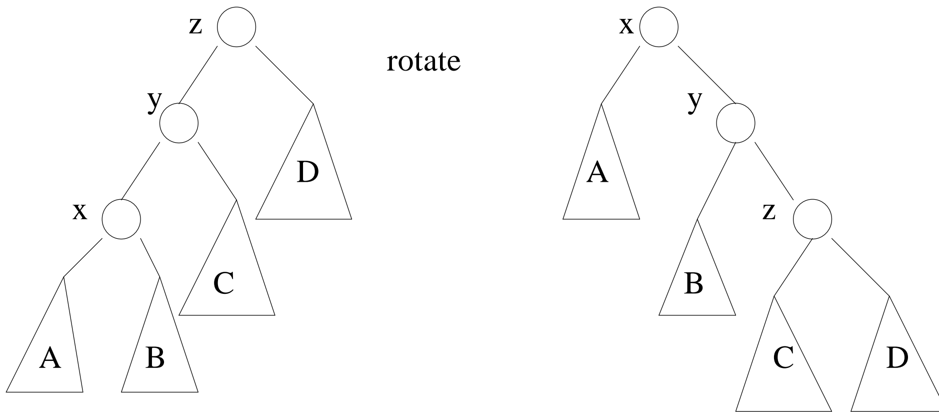
Case 3. Deletion occurred in D. We lost the height by one, and need to keep adjustment towards root.



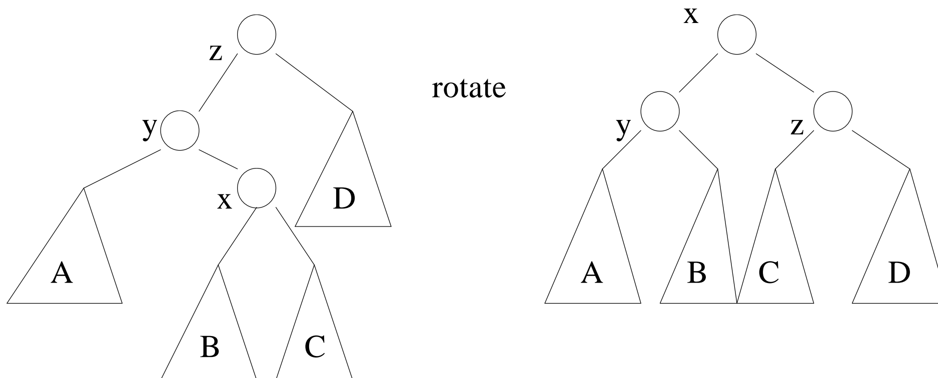
1.4 Splay Trees

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.

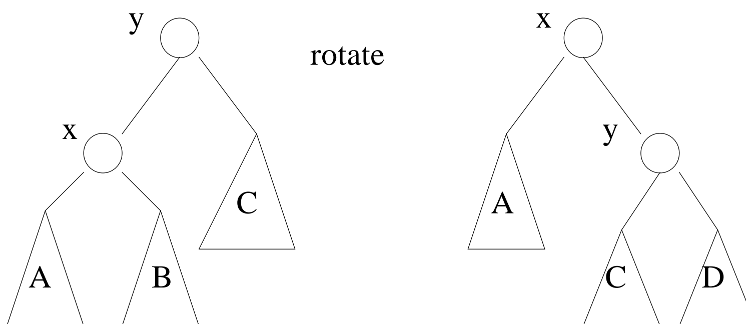
Case 1



Case 2



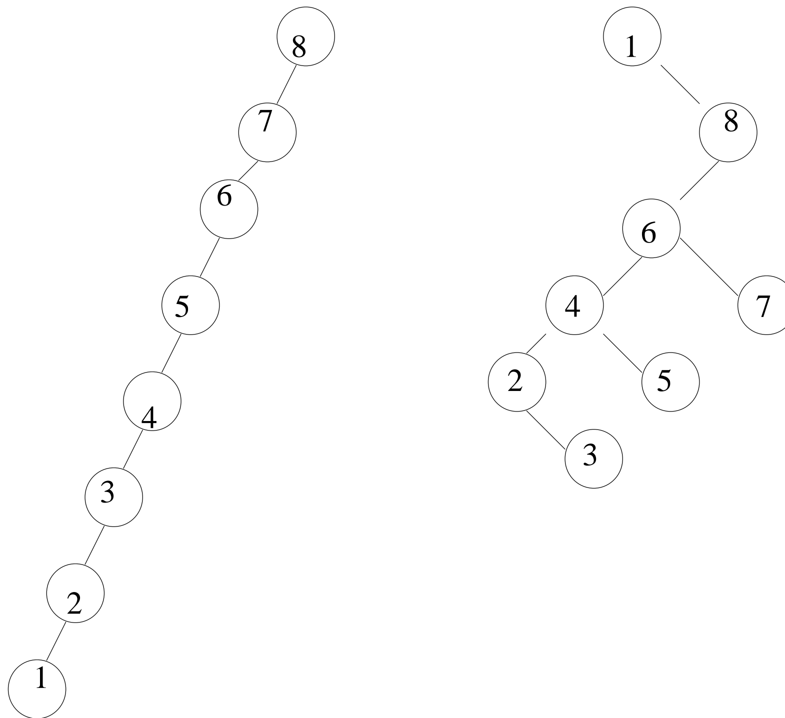
Case 3



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

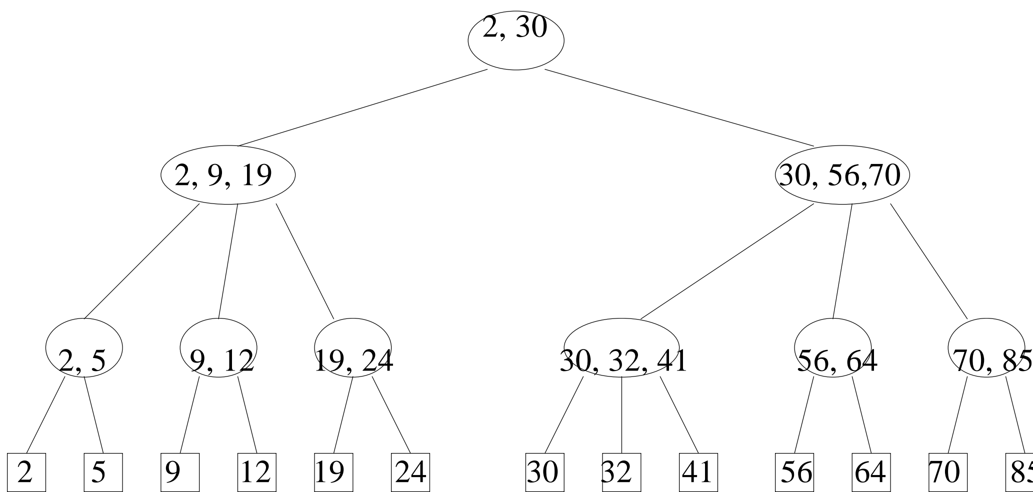
The concept of saving is defined by various means in various algorithms. In splay trees, amortised time is shown to be O(log n). The concept of amortised analysis is important when we perform many operations. It is known that m splay operations take O((m+n)log n) actual time through the amortised analysis..

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 ≤ x < key3 (if any), go to the second child.
5. If key3 ≤ 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.

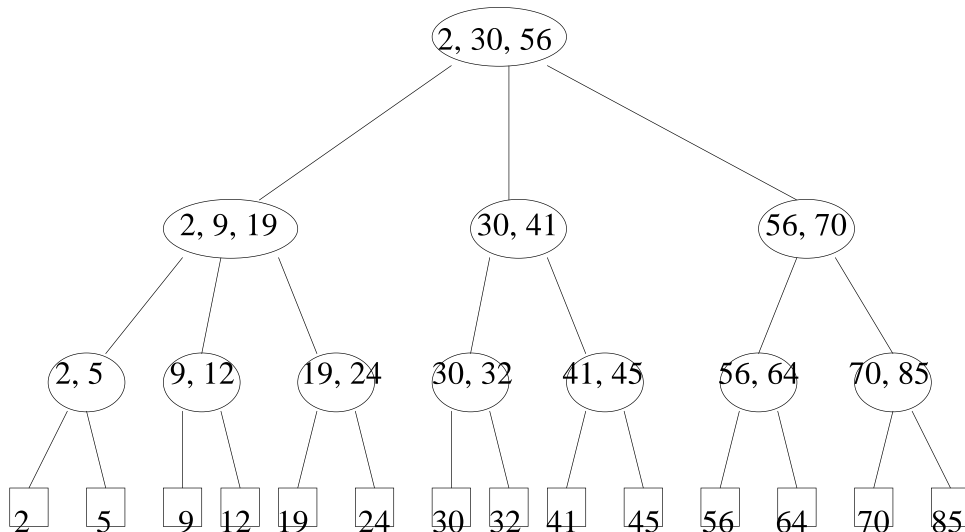
The height of the tree is O(log n) and find(x) takes O(log n) time. Insertion and deletion cause reorganisation of the tree.

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.

1.6 Heap ordered trees (or priority queues)

A heap ordered tree is a tree satisfying the following condition.

“The key of a node is not greater than that of each child if any”

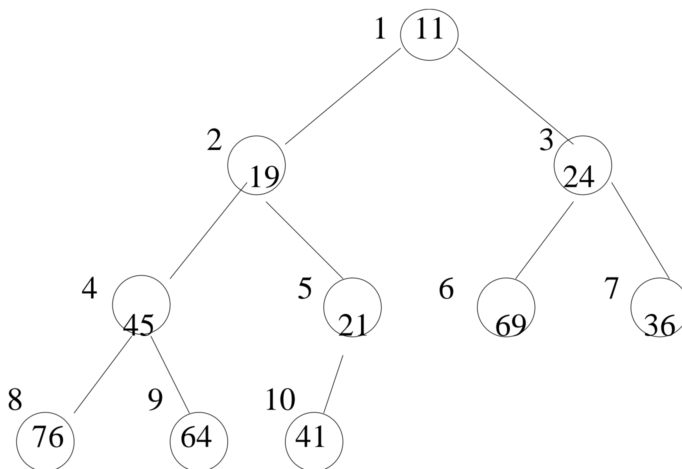
In a heap ordered tree, we can not implement “find” efficiently, taking O(n) time when the size of the tree is n. We implement the following operations efficiently on a heap ordered tree. Note that “find” is missing.

min
update_key (decrease_key, increase_key)
delete
insert
meld

1.6.1 **The heap**

The heap is a heap prdered tree of the complete binary tree shape. In general, a heap ordered tree needs a linked structure which requires some overhead time to traverse, whereas a heap can be implemented in an array on which heap opearations efficiently work.

Example



The labels attached to nodes represent the array indices when implemented on an array

11	19	24	45	21	69	36	76	64	41
1	2	3	4	5	6	7	8	9	10

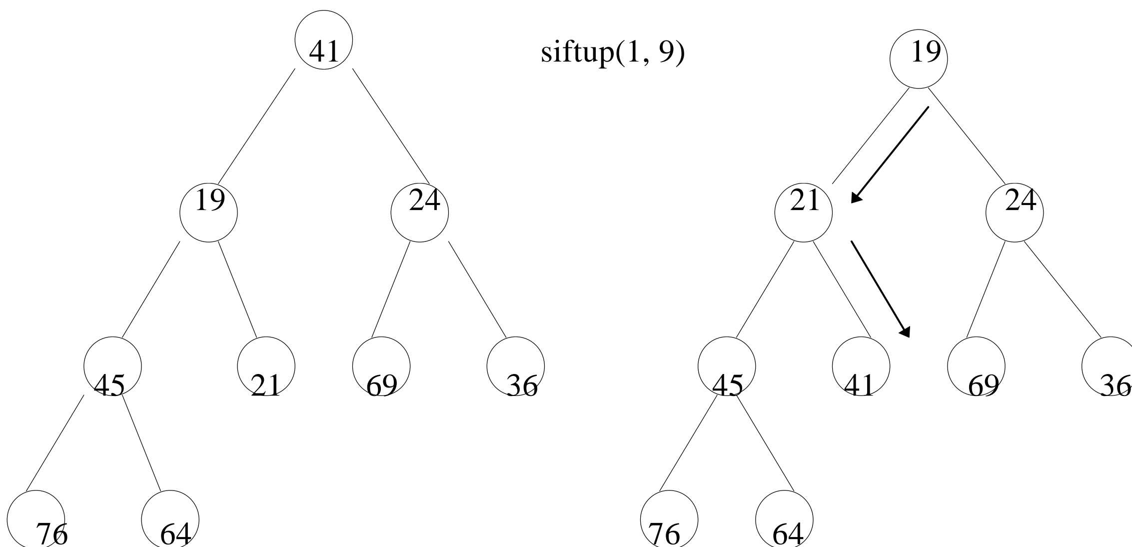
A complete binary tree occupies the first n elements in an array. If a node i (indicated by index i) has one or two children the left child is at 2i and the right child (if any) is at 2I+1. The parent of node j (j>1) is at j div 2. These opearations can be implemented by shift operations in machine code and thus fast.

Suppose the given heap is contained in a[1 .. n]. The procedure siftup(p, q) adjusts the subtree rooted at p so that the whole tree can be recovered to be a heap when only the key at p may be greater than those of its children and thus violates the heap condition. The parameter q indicates the end of the subtree.

```
1.  procedure siftup(p, q);
2.  begin
3.    y:=a[p]; j:=p; k:=2*p;
4.    while k <= q do begin
5.      z:=a[k];
6.      if k < q then if z > a[k+1] then begin k:=k+1; z:=a[k] end;
7.      if y <= z then goto 10;
8.      a[j]:=z; j:=k; k:=2*j
9.    end;
10.  10: a[j]:=y
11. end;
```

The item with a smaller key is repeatedly compared with y and go up the tree each one level until y finds its position and settles at line 10. O(log n) time.

Example. Pick up the key 41 at node 10 and put it at node 1. The key 41 comes down and we have the following.



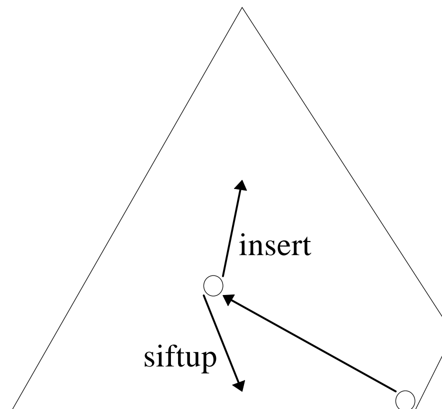
The procedure insert(x) is described below. O(log n) time.

```
1.  procedure insert(x);
2.  begin
3.    n:=n+1;
4.    i:=n;
5.    while i >= 2 do begin
6.      j := i div 2;
7.      y := a[j];
8.      if x >= y then go to 10;
9.      a[i] := y;  i:=j
10.  end;
11. 10: a[i] := x
12. end;
```

The key x is put at the end of heap and if the key value is small, we go up the path toward the root to find a suitable place for x.

The delete operation is a combination of insert and siftup operations. O(log n) time.

```
1.  {x=a[p] is to be deleted}
2.  begin
3.    n:=n-1;
4.    if p <= n then
5.      if a[p] <= a[n+1] then begin
6.        a[p] := a[n+1];
7.        siftup(p, n)
8.      end else begin
9.        m:=n; n:=p-1;
10.       insert(a[m+1]); n:=m
11.     end
12. end.
```

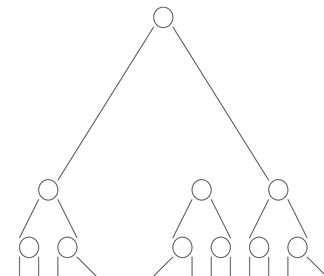


The “min” operation is easy. The minimum is at the root. O(1) time. The operation decrease_key(p) is similar to insert and increase_key(p) is similar to siftup, where p is the node index where the key value is changed. These take O(log n) time.

The “meld” opearation which melds two priority queues is not easy for the heap. This is easy for the binomial heap which will be described in the next section.

The opearation build_heap is to create a heap given in the following.

```
1.  procedure build_heap;
2.  begin
3.    for i:=n div 2 downto 1 do siftup(i, n)
4.  end.
```



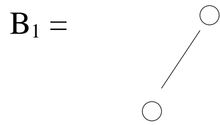
This procedure create a heap in a bottom-up manner.
The computing time can be shown to be O(n).

1.6.2 **Binomial Queues**

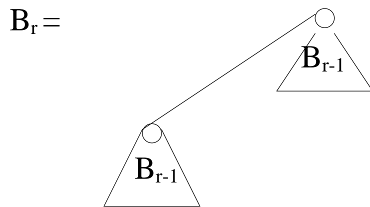
A binomial tree B_r is recursively defined as follows:

B_0 =

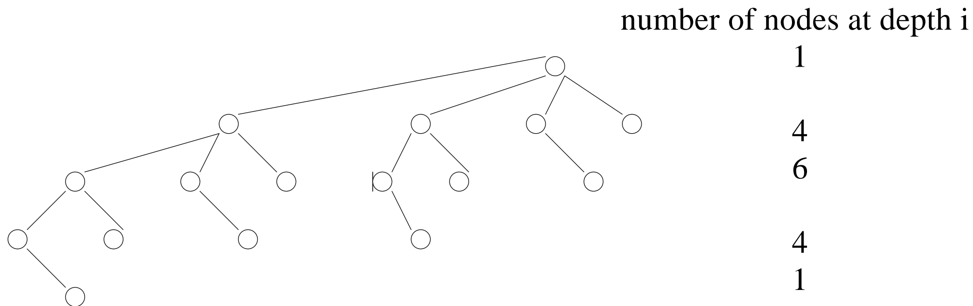
B_1 =



B_r =



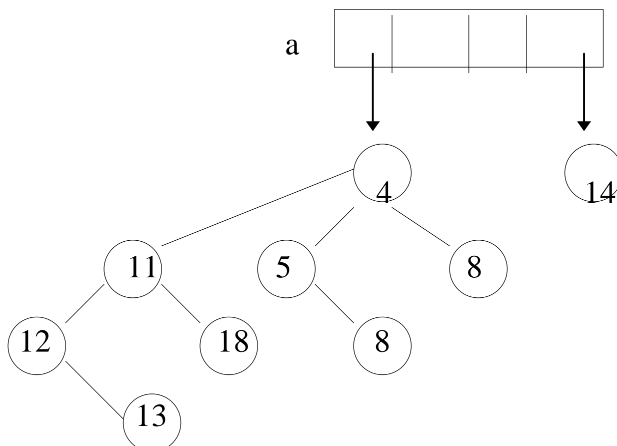
Example B_4



At depth i, there are C(r, i) nodes, where C(r, i) =r!/i! is a binomial coefficient, hence the name.

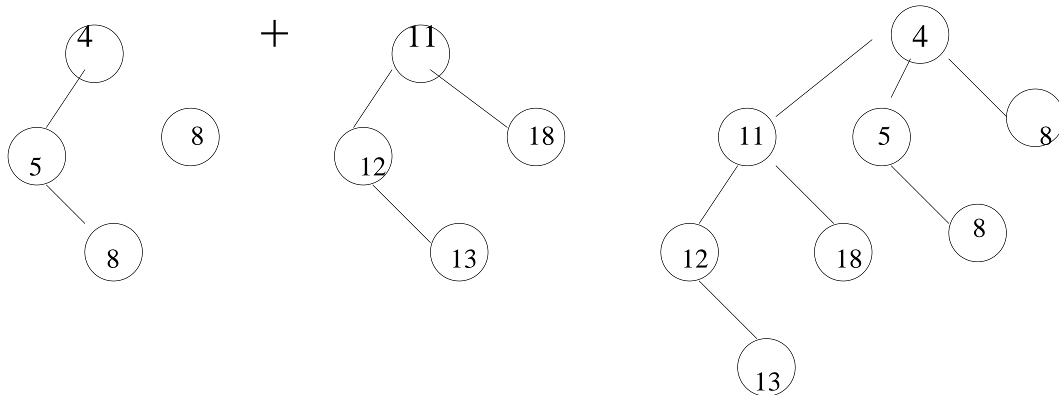
A binomial queue is a forest of binomial trees. The root of each tree in the forest is pointed to by an array element.

Example. n=9 in a binary form is $(1001)_2$. Then the binomial queue consits of B_3 and B_0 as shown below.



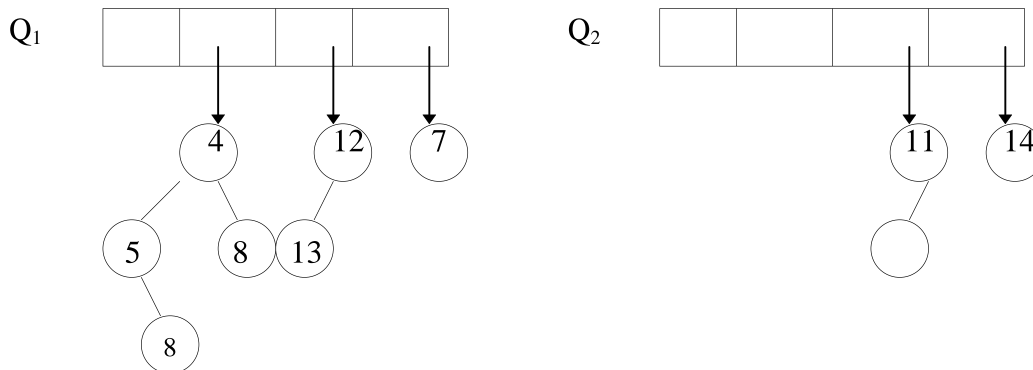
Note that the size m of array a is m = $\lfloor \log_2 n + 1 \rfloor$. There is no ordering condition on the keys in the roots of the binomial trees. Obviously the minimum is found in O(log n) time. The melding of two binomial trees can be done in O(1) time by just comparing the roots and linking them.

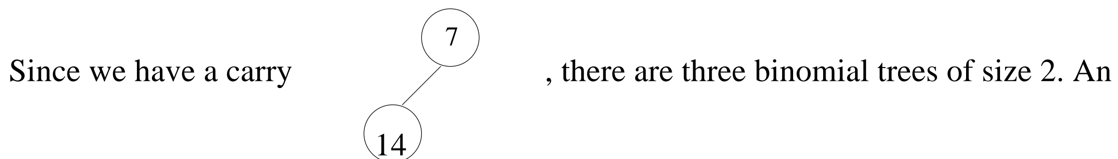
Example

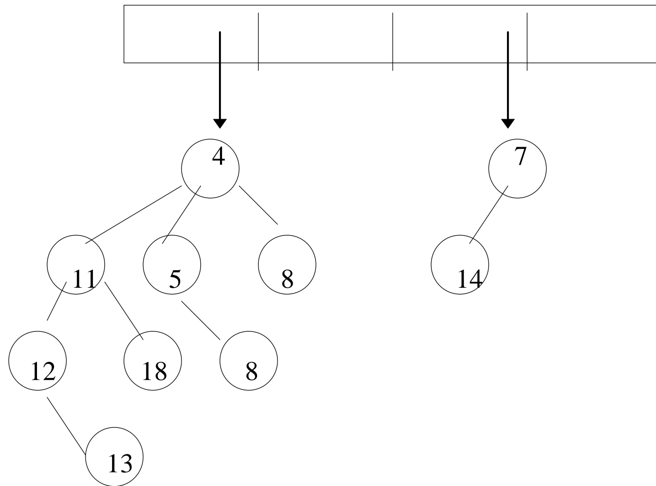


The meld operation for two binomial queues can be implemented like a binary addition. Let the two binomial queues to be melded be Q_1 and Q_2. We proceed with melding the 0-th binomial trees , first binomial trees, second binomial trees, and so forth, with carries to higher positions.

Example 7 = $(0111)_2$ and 3 = $(0011)_2$. Then 10 = $(1010)_2$.



Since we have a carry  , there are three binomial trees of size 2. An arbitrary one can be put at position 1 and the other two are melded and carried over to the next position.. Now there are two binomial trees of size 4, which are melded to produce a carry to the next position. Finally we have the following.



The time for meld is O(log n).

The operation “delete_min” is described as follows: Perform find_min first. Deleting it will cause the tree containing it to be dispersed into fragments, resulting in another binomial queue. Now meld the original binomial queue from which the tree containing the minimum was removed with the new binomial queue. Time is O(log n).

The operations decrease_key and increse_key can be implemented in O(log n) time, since we can repeatedly swap the element updated with ancestors or descendants, and the longest path length is O(log n).

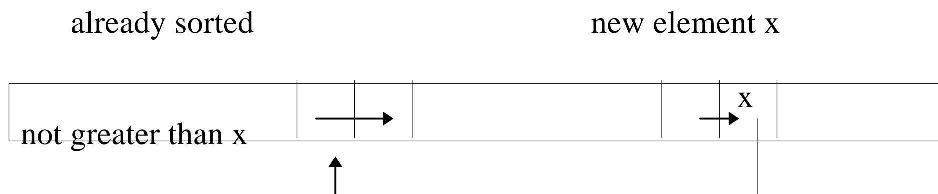
The Fibonacci heap is a variation of the binomial queue, invented by Fredman and Tarjan in 1984. The remarkable aspect of this data structure is that we can perform n delete_min and m decrease_key operations in O(m + n log n) time.

2. Sorting

Sorting is to sort items with their keys in increasing order. For simplicity we only consider keys to be rearranged. Computing time will be measured mainly by the number of comparisons between keys.

2. 1 Insertion sort

This method works on an one-dimensional array (or simply array). A new element is inserted into the already sorted list by shifting the elements to the right until we find a suitable position for the new element. See below.



This method takes about $n^2/4$ comparisons and not very efficient except for small n.

2.2 Selection sort

This method repeatedly selects the minimum from the remaining list. If we implement this method only using an array, we need n-1 comparisons for the minimum, n-2 comparisons fro the second minimum, and so forth, taking about $n^2/2$ comparisons, not efficient.

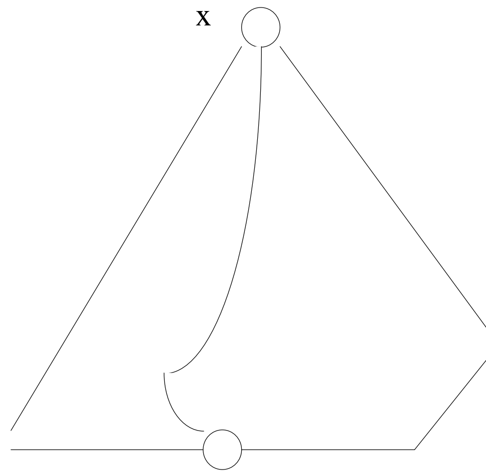
We use a priority queue to select successive minima. The abstract form of algorithm is described as follows:

1. Construct a priority queue
2. Initialise the list for the sorted result
3. Repeatedly select and delete the minimum from the queue and append it to the list

The most efficient selection sort is heapsort invented by Williams and Floyd in 1964.

1. build_heap;
2. **for** i:=n downto 2 **do begin**
3. swap(a[1], a[i]);
4. siftup(1,i-1)
5. **end.**

Since we need two comparisons to come down the heap after the swap operation, the number of comparisons for one siftup is bounded by 2log(n). Hence the computing time is O(nlog n). For random keys, the swapped element x put at the root is likely to come down near the bottom of the tree. Regard the root as a hole. A small gain can be made here by sending the smaller child up along the path towards the bottom, and the finding a suitable position for x by going up from the bottom. We spend about log n comparisons to come to the bottom, and a few more to go up. See below.

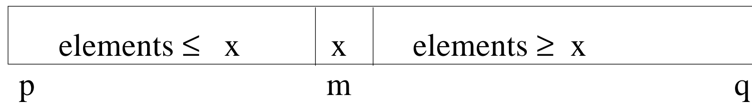


2.3 Exchange sort

Repeatedly exchange two elements until sorted. The most efficient and famous exchange sort is quicksort invented by Hoare in 1961, described below.

```
1.  procedure quicksort(p, q);
2.  begin  m : local
3.    if p<q then begin
4.       x:=a[p];
5.       partition(p, q, m); /* this procedure should be expanded here from next page */
6.       quicksort(p, m-1);
7.       quicksort(m+1, q);
8.    end
9.  end;
10. begin {main program}
11.   quicksort(1, n)
12. end.
```

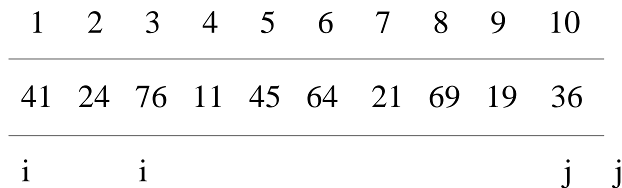
partition(p, q, m) is to put all elements not greater than x in a[p .. m-1], those not smaller than x in a[m+1 .. q] and x at a[m]. See below.

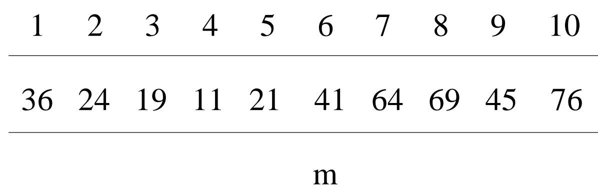


The following procedure for partition is more efficient than the original version made by Hoare, invented by Takaoka in 1984.

```
1.  procedure partition(p, q, m);
2.  begin
3.    i:=p; j:=q+1;
4.    while  i < j do begin
5.        repeat
6.          j:= j-1;
7.         if  i = j then goto 10
8.        until a[j] < x;   /* this < can be ≤ */
9.        a[i]:= a[j];
10.       repeat
11.         i:= i+1;
12.         if  i = j then goto 10
13.       until a[i] > x;  /* this > csn be ≥  */
14.       a[j]:= a[i]
15      end;
16.  10: a[i] := x; m:= i
17. end;
```

The parameter m should be of var type.The behaviour of partition can be depicted by the following.





j stops at 10 and 36 is put at 1. Next i stops at 3 and 76 is put at 10, etc. This procedure is particularly efficient when there are many identical elements. When we find x = a[i] or x = a[j], we just keep going within the inner loop, whereas swapping takes place in this situation in the original partition by Hoare.

Now let T(n) be the number of comparisons between the input data, not including those between control variables. Then we have the following recurrence.

$T(1) = 0$

$T(n) = n - 1 + (1/n)\sum_{[i=1 \text{ to } n]}(T(i-1) + T(n-i)).$

We assume that x falls on the range 1 .. n with equal probability 1/n. Partition above takes n-1 comparisons whereas Hoare's takes n+1 comparisons in the worst case. Similarly to the analysis for binary search trees, we have T(n) is nearly equal to $1.39n\log_2 n$.

Ironically the worst case of quicksort occurs when the given list is already sorted, taking $O(n^2)$ time. This happens because we choose the value of x at line 4 in procedure quicksort, which is called the pivot, to be the leftmost element. To prevent the above mentioned worst case, we can take the median of a[p], a[(p+q) div 2] and a[q]. It is known that if the pivot comes near to the center, the performance is better. The above choice also contribute in this aspect. There are many more strategies on how to choose good pivots.

2.4 Merge sort

Mergesort sorts the list by repeatedly merging the sublists from the single elements at the beginning until there is only one sorted list.

Example

41	24	76	11	45	64	21	69	stage 0

24 41	11 76	45 64	21 69	stage 1

11 24 41 76	21 45 64 69	stage 2

11 21 24 41 45 64 69 76	stage 3

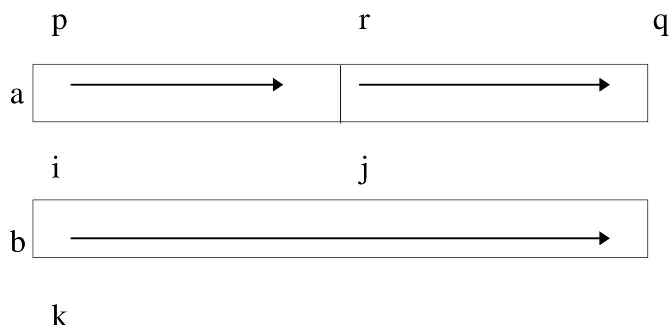
The procedure mergesort is given below.

```
1.  procedure mergesort(p, q);
2.  begin  m : local
3.    if p < q then begin
4.      m := (p+q) div 2;
5.      mergesort(p, m);
6.      mergesort(m+1, q);
7.      merge(p, m+1, q+1)
8.    end
9   end;
9.  begin { main program }
10.   mergesort(1. n)
11. end.
```

procedure merge(p, m, q) is to merge sorted arrays a[p .. r-1] and a[r .. q-1] described below.

```
1.  procedure merge(p, r, q);
2.  begin
3.    i:=p; j:=r; k:=p;
4.    while i < r and j < q do begin
5.      if a[i] <= a[j] then begin b[k] := a[i]; i:=i+1 end
6.                         else begin b[k] := a[j]; j:=j+1 end;
7.      k:=k+1
8.    end;
9.    while i < r do begin
10.     b[k] := a[i];
11.     i:=i+1; k:=k+1
12.   end;
13.   while j < q do begin
14.     b[k] := a[j];
15.     j:=j+1; k:=k+1
16.   end;
17.   for k:=p to q-1 do a[k] := b[k]
18. end;
```

The situation is illustrated in the following.



In the worst case, merge takes m+n-1 comparisons between array elements where m and n are the lengths of the merged lists. See below. The first m-1 elements in the left list is smaller than the first of the right. The last in the left is greater than any in the left.

m n

n comparisons m-1 comparisons

Let $n = 2^k$ for some k, and T(n) be the number of comparisons for mergesort. Then we have the recurrence

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

Solving this recurrence yields $T(n) = n\log_2 n - n + 1$. In terms of comparisons, mergesort is one of best. The drawback is that we need an extra array b, and also the number of assignment statements executed is greater than that of other methods.

Other merits of mergesort is that it is suitable for external sort which uses external is briefly described below.

Natural mergesort scans the list from left to right to find ascending runs spending O(n) time, and then repeatedly merges two adjacent runs. See below.

k runs

stage 0

stage 1

stage 2

After $\log_2 k$ stages, we are done. At each stage we consume at most n-1 comparisons. Thus the computing time O(nlogk), precisely speaking O(nlog(k+1)) to prevent T(n)=O(0) when k=1, in which case T(n)=O(n).

Minimal mergesort, invented by Takaoka in 1996, repeatedly merges two shortest remaining runs after the prescanning. Let $p_i = n_i / n$ where n_i is the length of the i-th run. Let entropy H(n) be defined by

$$H(n) = - \Sigma \text{ (i=1 to k) } p_i \log p_i$$

Then minimal mergesort sorts the given list in O(nH(n)) time, or precisely speaking O(n(H(n)+1). Note that $0 < H(n) < \log k$.

Example $n_1 = n_2 = 1$. $n_i = 2n_{i-1}$ for $i > 2$. The natural mergesort takes O(n loglog n) time, whereas minimal mergesort takes O(n) time.

2.5 Radix sort

Radix sort is totally different from other methods. It does not compare the keys, but inspects the digits of the given numbers, hence the name There are two ways of inspecting the digits. One is to start from the most significant digit (MSD)and the other from the least significant digit (LSD).

Let us proceed with examples.

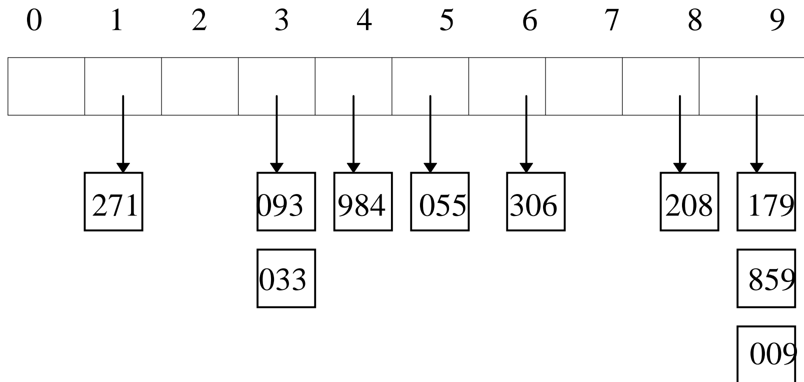
(179, 208, 306, 093, 859, 984, 055, 009, 271, 033)

The MSD method classifies these ten numbers, starting at the first digit into

(093, 055, 009, 033), (179), (208, 271), (306), (859), (984),

in order of 0 .. 9. Empty lists are omitted. We apply the same procedure to each non-empty list for the second digit, third, and so forth. This method is conceptually simple, but not efficient for actual implementation.

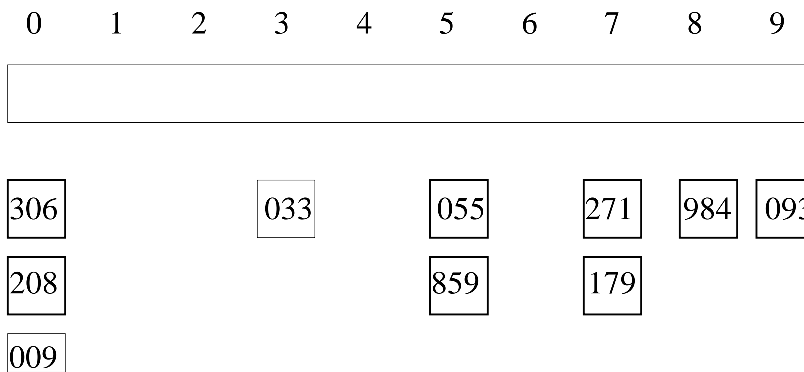
Next we describe the LSD method. We start at the least significant digit. As we scan the list from left to right, we put the numbers in linearly ordered list each linked from the i-th array element if the least significant digit is I. The linearly ordered lists are sometimes called buckets.



The we traverse the lists from left to right and in each list from up to down, and we concatenate them into a single list as follows:

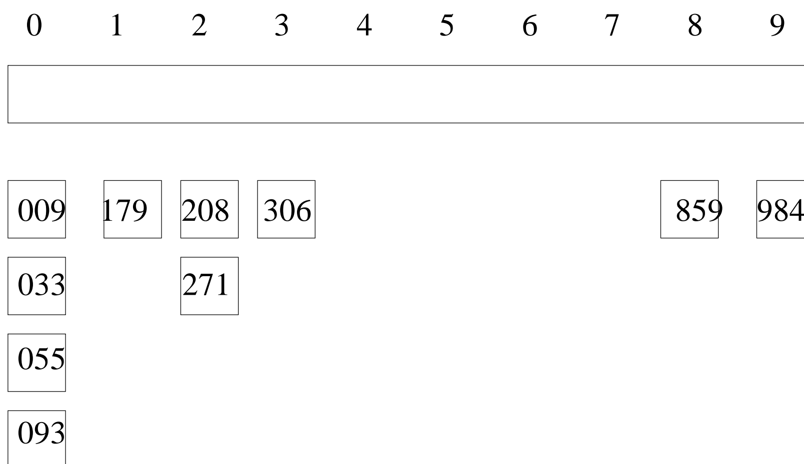
(271, 093, 033, 984, 055, 306, 208, 179, 859, 009)

We do the same using the second least significant digit as follows:



(306, 208, 009, 039, 055, 859, 271, 179, 984, 093)

Finally we have



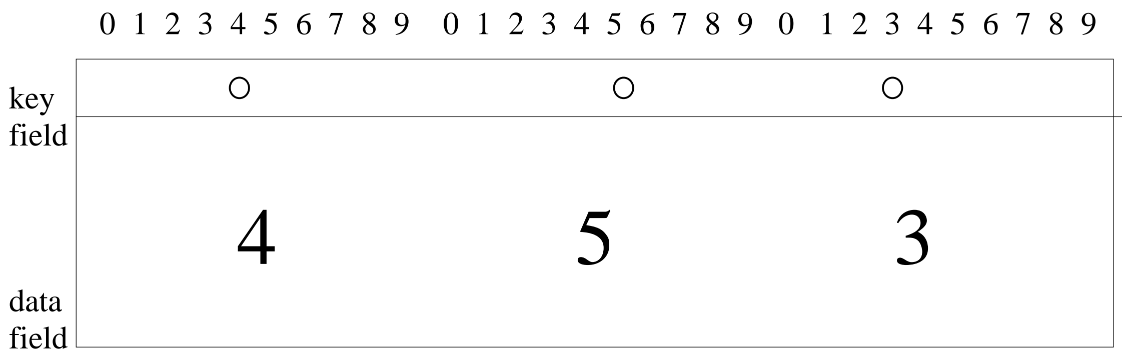
(009, 033, 055, 093, 179, 208, 271, 306, 859, 984)

At the end of i-th stage, numbers are sorted for the last i digits. Computing time is O(dn) where d is the number of digits. The method can be generalized to radix-r number. That is, each number x is contained in an array and expressed as a radix-r number as follows:

$$x = x[d-1]\, r^{d-1} + \ldots + x[1]\, r + x[0].$$

The computing time then becomes O(d(n+r)). In the previous example, d=3 and r=10. For fixed d and r, this is O(n). When n is large as compared to d and r, this method is very efficient, but when n is small, there will be many empty buckets and time will be wasted checking empty buckets..

This method is also suitable for manual sorting with cards as described below.



In the above, the key 453 is punched, shown by little circles. The other positions are punched out as shown below. Imagine we have a few hundred cards. By a stick we can pick up cards with keys whose last digits are 3 when we insert it at digit 3 in the last subfield. We do this for 0 .. 9 in the last subfield and put the cards in this order.. Next we do the same for the subfield 2, and so on. The IBM Company made this machine in the ancient times.