18.1-1
Why don't we allow a minimum degree of $t = 1$?

Step 1 of 1

According to the definition of B-Tree, there are lower and upper bounds on the number of keys a node can have. These bounds are represented using the minimum degree *t*, where $t \geq 2$.

• Except the root node, all other nodes should have minimum $t-1$ keys.

• Every internal node except the root will have minimum of *t* children.

• In a nonempty tree, the root node must have minimum one key.

• Every node can have maximum $2t-1$ keys.

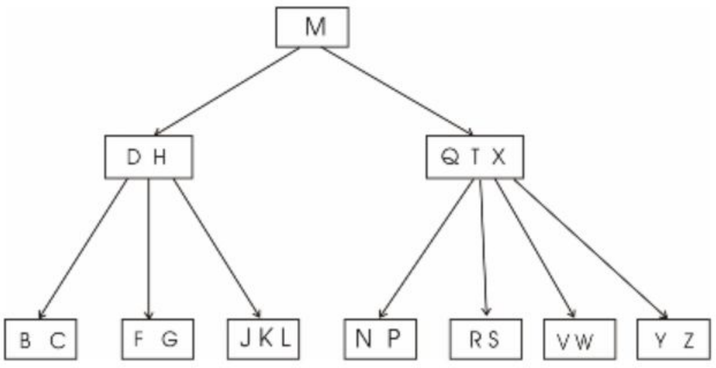
• Every internal node can have maximum of 2*t* children.

If the degree of *t* is 1, the minimum number of nodes will be 0 and the B-tree will be an empty tree. A nonempty B - tree should contain at least one key, thus making degree $t = 2$.

Hence the minimum degree of $t = 2$ and the minimum degree of $t = 1$ is not allowed.

18.1-2
For what values of t is the tree of Figure 18.1 a legal B-tree?

Step 1 of 2



Step 2 of 2

According to property 5

a. Every node other than root must have at least $t-1$ keys.

b. Every node can contain $2t-1$ keys.

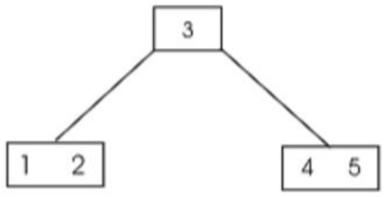
If we take $t = 2$ then according to (a) $t-1$ i.e., 1 which is not satisfying the property (a). Then take $t = 3$, then $t-1 = 2$ is minimum of keys and $2t-1 = 5$ is the maximum number of keys.

18.1-3
Show all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$.

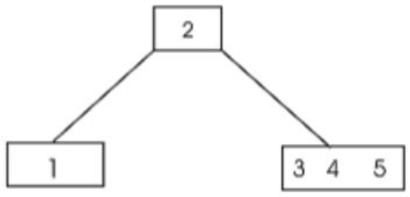
Step 1 of 1

The all legal B-trees of minimum degree 2 that represent {1, 2, 3, 4, 5} is shown below.

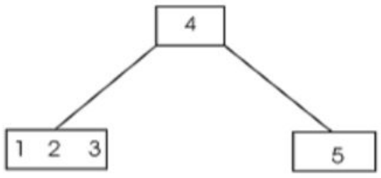
(i)



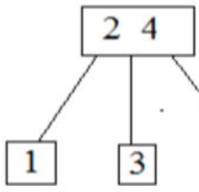
(ii)



(iii)



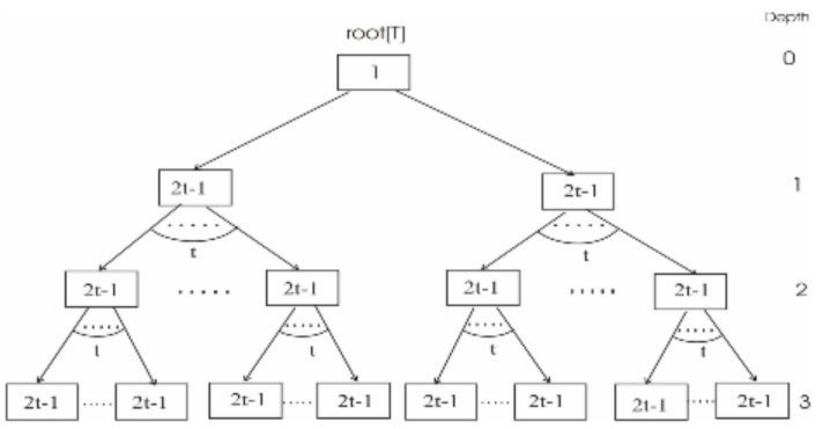
(iv)



18.1-4

As a function of the minimum degree t, what is the maximum number of keys that can be stored in a B-tree of height h?

Step 1 of 2



Depth	Number of nodes	Keys
0	$2t^0 = 1$	2t-1
1	$2t^1$	2t(2t-1)
2	$2t^2$	$2t^2(2t-1)$
3	$2t^3$	$2t^3(2t-1)$

A B – tree of height 3 containing a maximum possible number of keys.

Step 2 of 2

Continuing the series, we see that we have the summation of the nodes times the maximum keys per node $= \boxed{(2t-1)\sum_{i=0}^{h} 2t^i}$

18.1-5

Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

Step 1 of 5

Fig: Red Black Tree

Step 2 of 5

B-Tree is a tree data structure in which each node can have more than two children and special attributes that is it stores the number of keys currently in node with the keys themselves. It also contains a boolean field which to confirm that if a node is internal node or the leaf node.

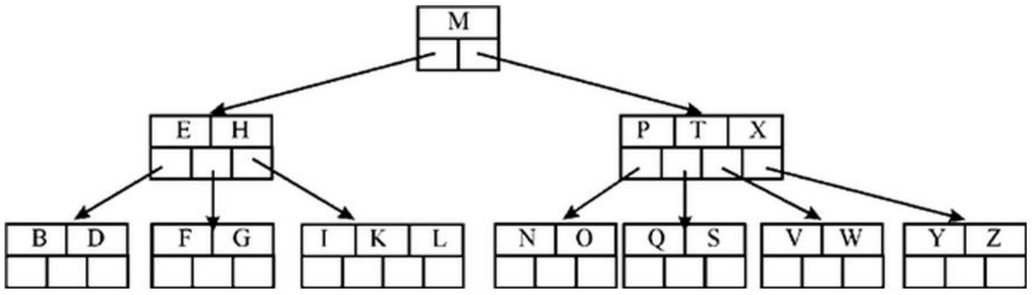


Fig: B-Tree

As the root node in a Red Black tree is black, we can say that if a black node would absorb its red children and incorporate them with its black children, we would get a new data structure.

Step 3 of 5

This data structure would be a generalized version of Binary search tree a tree with features similar to B Tree. It is because if all the black nodes would absorb its red children and make their children as its own, then the red black tree's height will reduce and there will be many nodes in a single level. As per the definitions of B-Tree, this description suits well. Hence we will get a B-Tree.

Step 4 of 5

Hence in a Red Black Tree if the black node absorbs its red children and incorporate its own children in that place then new formed data structure will be B-Tree.

Step 5 of 5

Supposed condition:

If the black node assimilate the red children and incorporate its own black children at that place in the Red Black Tree then new data structure forms.

Now it is finding that, data structure is the incorporation of black nodes in place of the red children. Red Black Tree is a special type of tree in which each node contains an extra bit for the storage of the color of that node.

So each node of Red Black Tree contains five fields *p* for parent, *left* for left child, *right* for right child, *key* for data value and *color* for color of the node.

The properties of Red Black Tree are as follows:

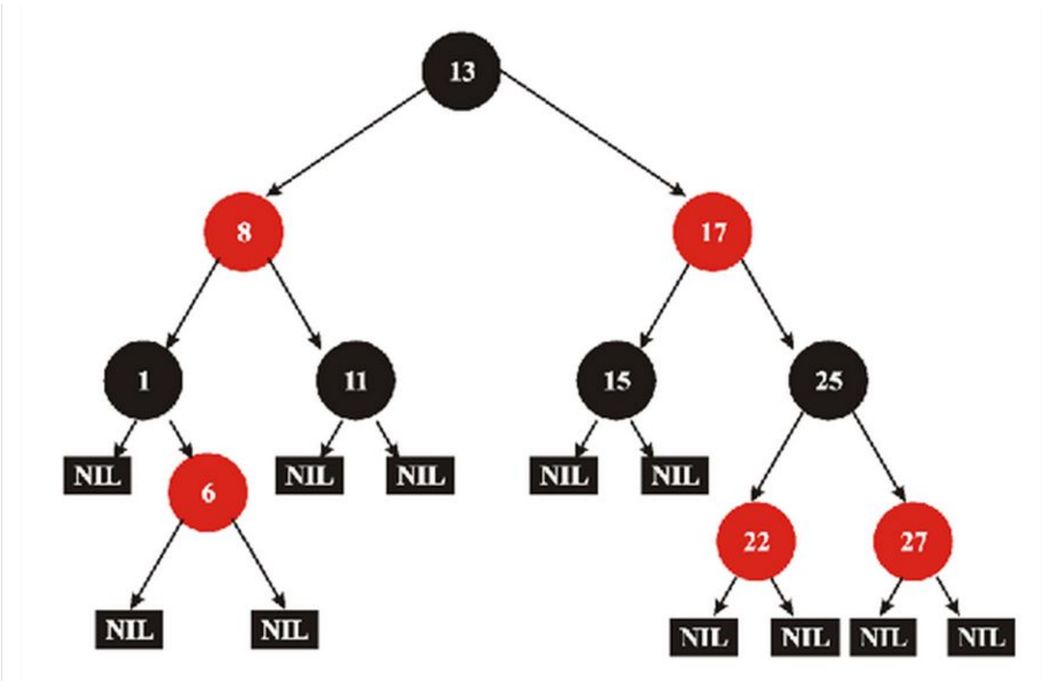
1. In this tree each and every node is either painted with red color or painted with black color.

2. The root node of the Red Black tree is always black. (This rule is not right in every condition. Since the root can always be changed from red to black, but not true for the change of color from black to red always)

3. All leaf nodes (NIL) are painted in black color.

4. For each red node its children are always black.

5. In the Red Black Tree each path from a node to its leaf contains equal number of black nodes.



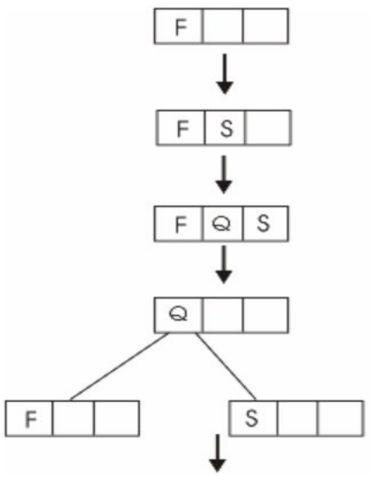
18.2-1

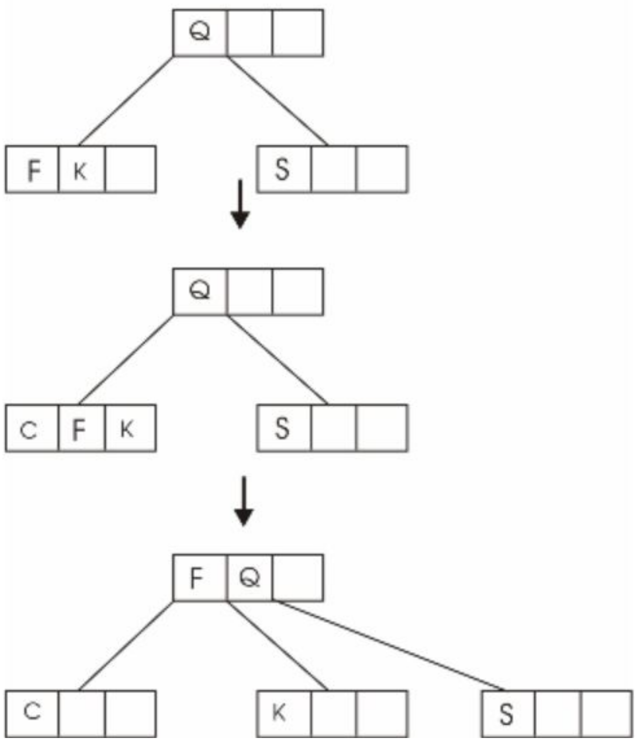
Show the results of inserting the keys

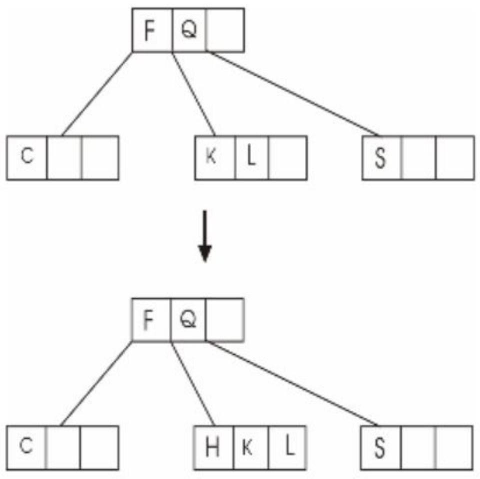
$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

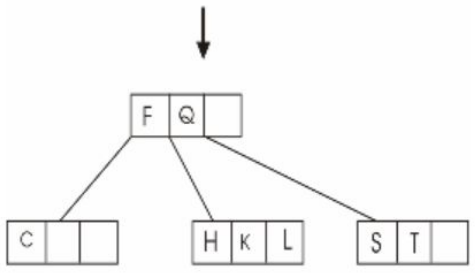
in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

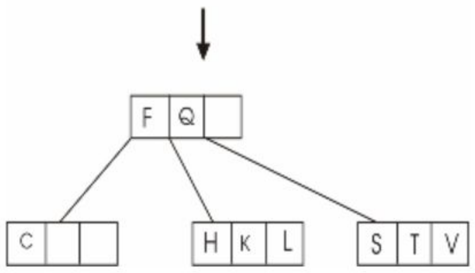
Step 1 of 11



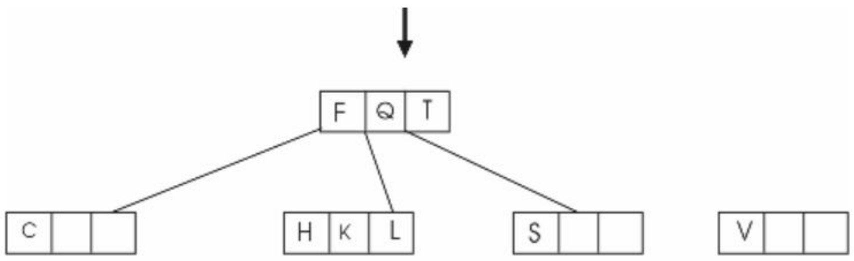


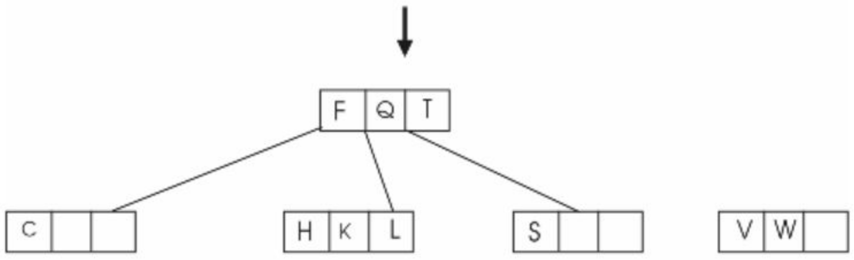


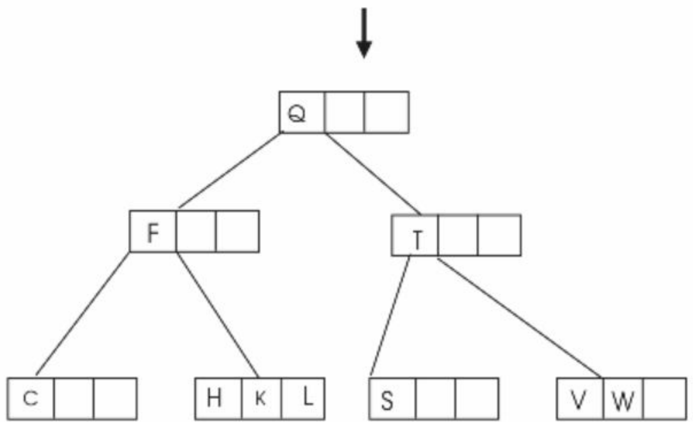




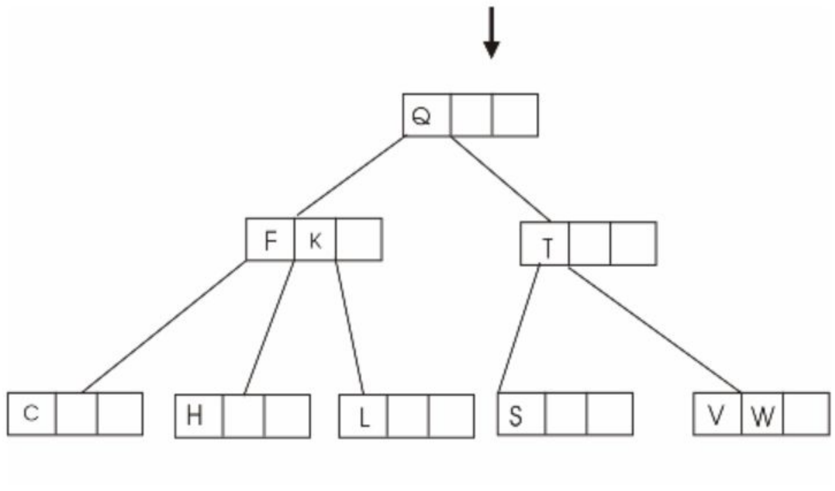
Step 3 of 11

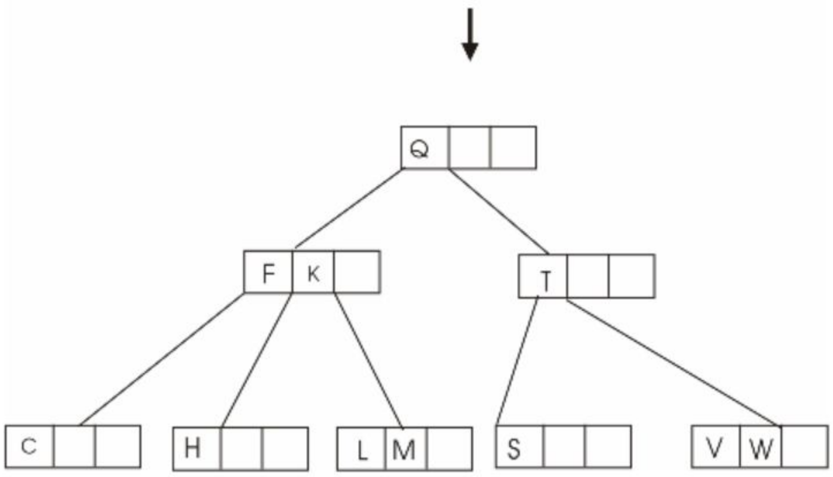




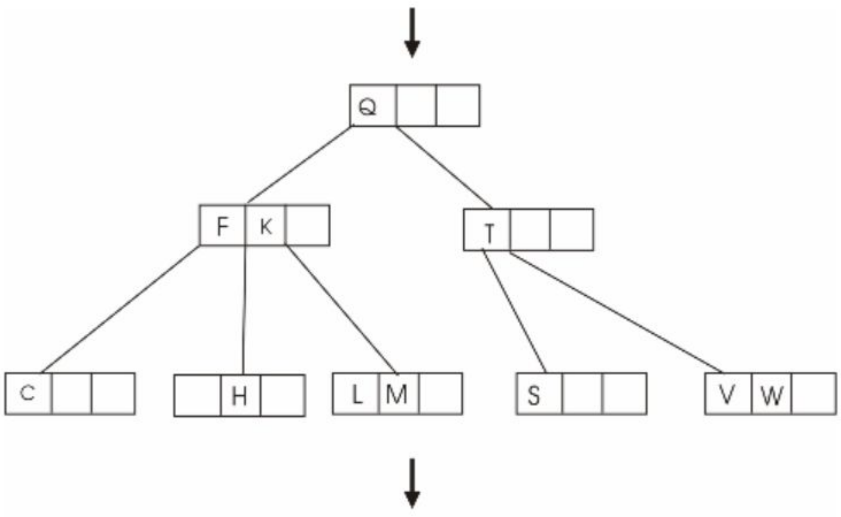


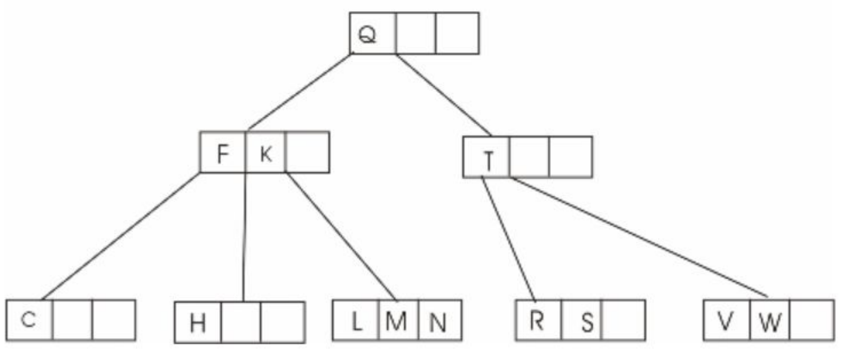
Step **4** of 11



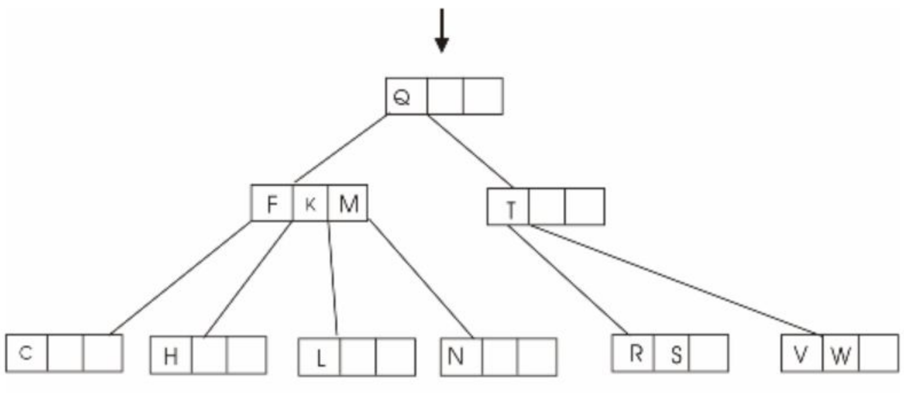


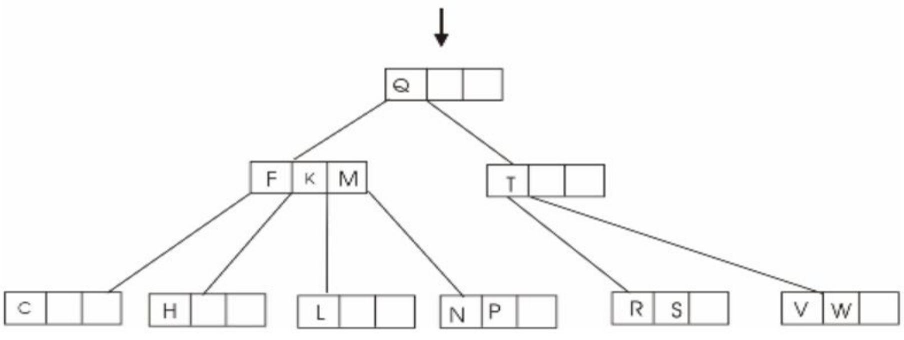
Step **5** of 11



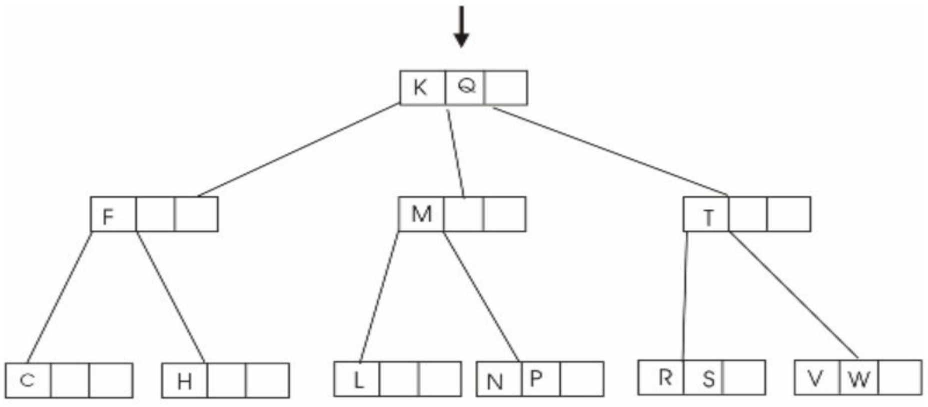


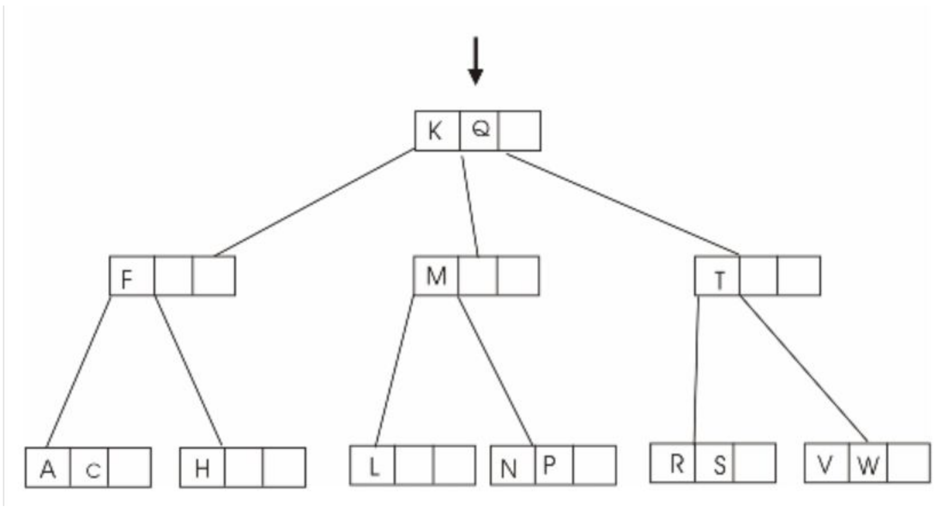
Step 6 of 11



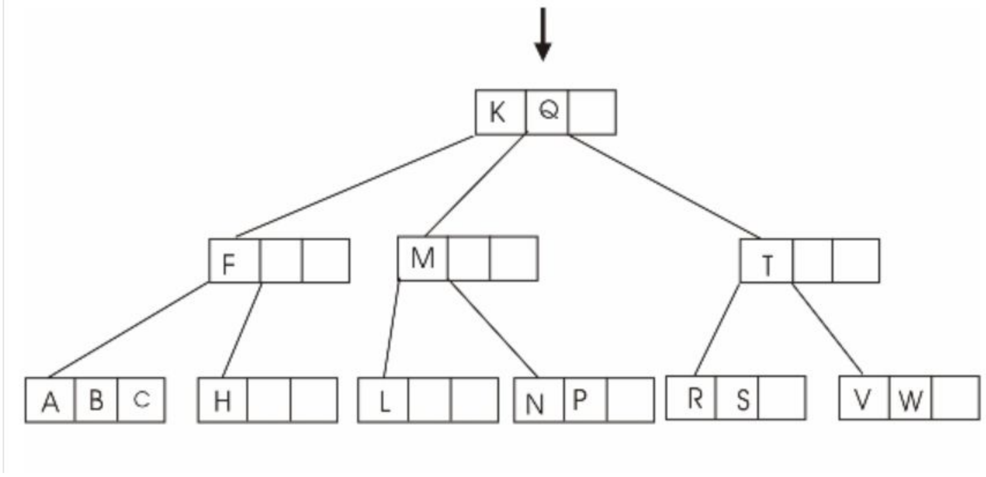


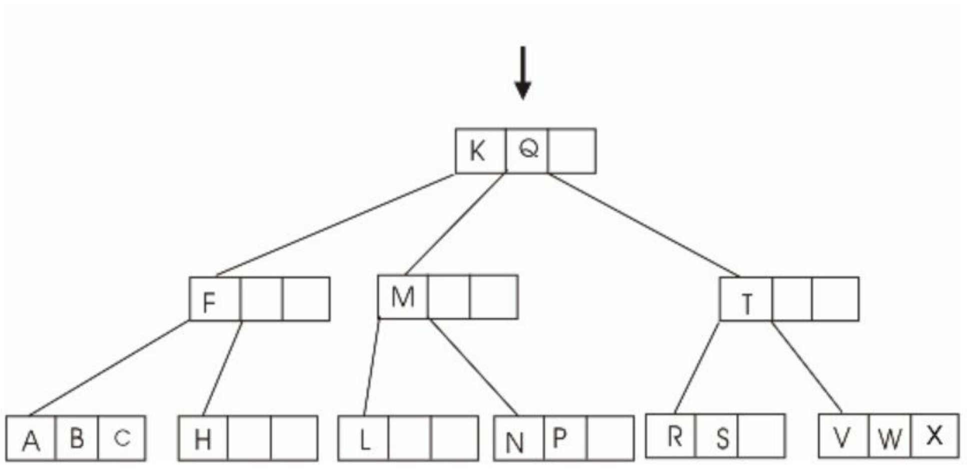
Step 7 of 11



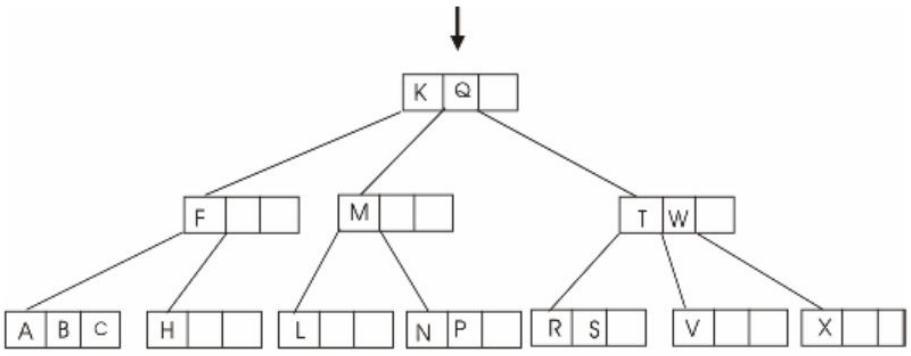


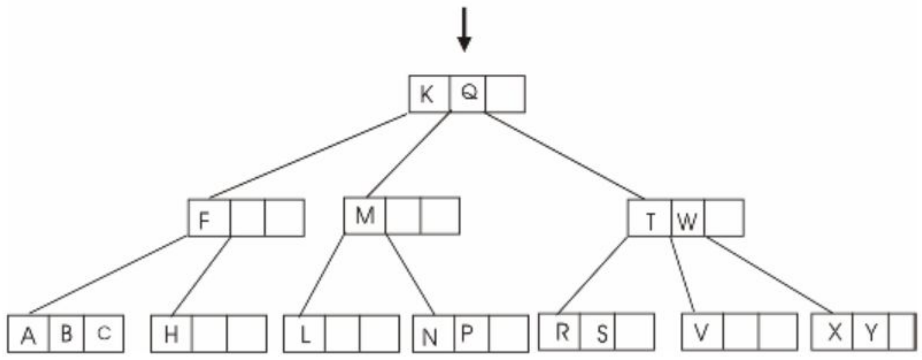
Step 8 of 11



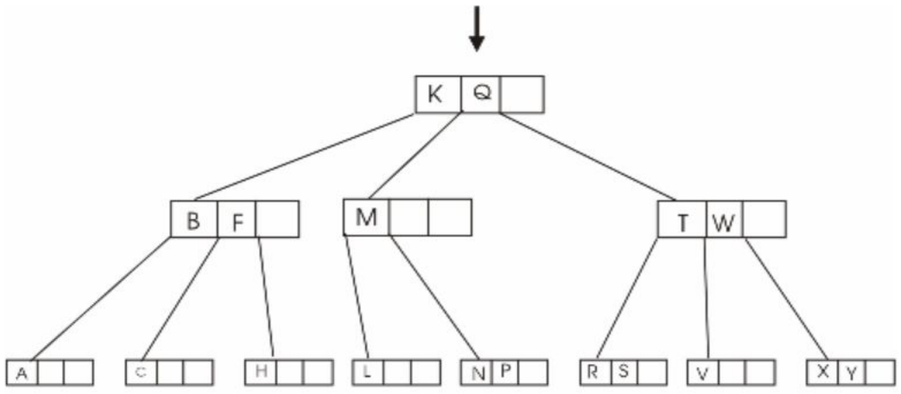


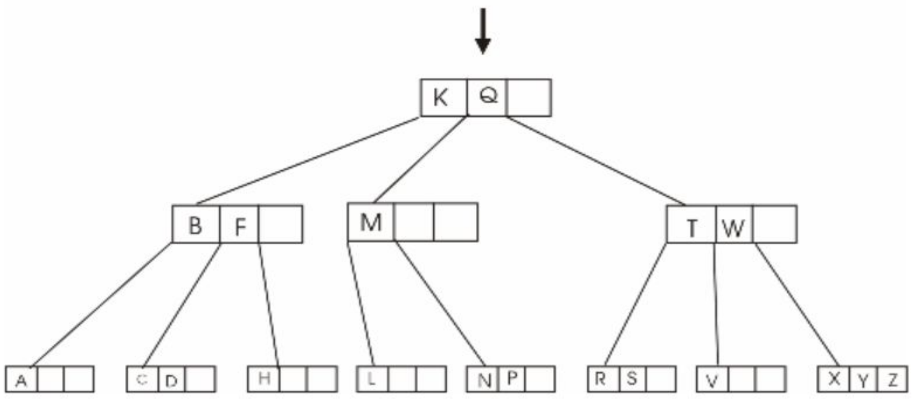
Step 9 of 11



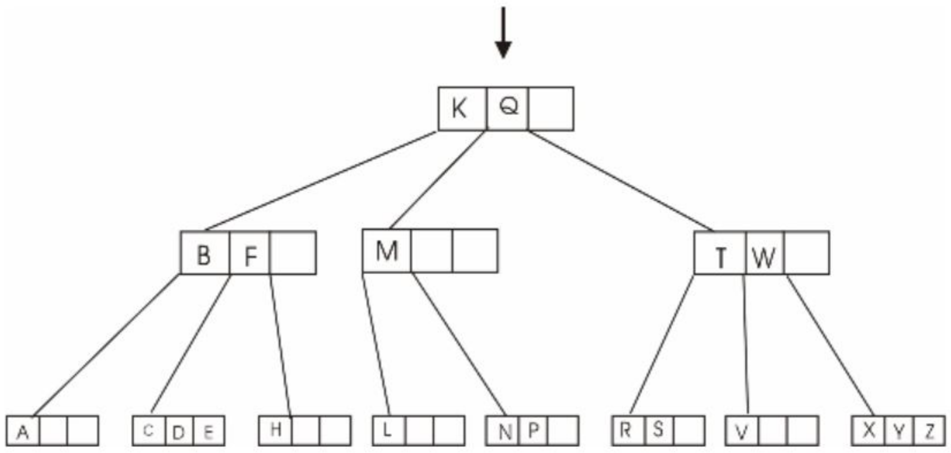


Step 10 of 11





Step **11** of 11



18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

Step **1** of 1

The circumstances of redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT is if the size of the tree is h and if there are full nodes in the h-1 level, one node will be written twice while inserting a key to the B-Tree. DISK-READ operation will be called twice; once while splitting the full node and once finally inserting the key.B-TREE-INSERT-NONFULL is being called with the initial nodes in memory and always READ the next level (child) nodes. Therefore, DISK-READ will occur only once for any node while inserting a key.

18.2-3

Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

Step **1** of 2

The procedure for maximum key stored in a B-tree is:

```
B-TREE-MAXIMUM(x)
1.i = n[x]
2.while not x.leaf
3. DISK-READ (x.c_i)
4. x = x.c_i
5. i = x.n
6. return x.key_i
```

Step 2 of 2

The procedure for the predecessor of a given key stored in a B-tree is:

```
B-TREE-PREDECESSOR(x,i) // x.key_i = k
1.if x.c_i! = NIL
2. then return B-TREE-MAXIMUM (x.c_i)
3. else if i > 1
4. then return x.key_{i-1}
5. else // when k is a leaf and the leftmost key
6. y =x. p
7. while y! = NIL and y.p.c_1 = y
8. y =y. p
9. j = 1
10 while y.c_j! = y
11. j = j + 1
12. if j = 1 then return NIL // no predecessor
13 else return y.key_{j-1}
```

18.2-4 ★

Suppose that we insert the keys $\{1, 2, \ldots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

Step 1 of 1

B-Tree

For a B-tree

• Except the root node, all other nodes should have minimum $t-1$ keys.

• Every internal node except the root will have minimum of *t* children.

• In a nonempty tree, the root node must have minimum one key.

• Every node can have maximum $2t-1$ keys.

• Every internal node can have maximum of 2*t* children.

• All leaves will have the same depth, which is considered as the height of the tree.

If the degree of *t* is 1, the minimum number of nodes will be 0 and the B-tree will be an empty tree. A nonempty B - tree should contain at least one key, thus making degree $t = 2$.

If *h* is the height of the B-tree, then the root contains minimum one key and all other nodes will contain $t-1$ keys.

• At depth 1 minimum 2 nodes

• At depth 2 minimum 2*t* nodes

• At depth 3 minimum $2t^2$ nodes

.

.

.

• At depth h minimum $2t^{h-1}$ nodes

$$\text{Summing up all the nodes } = 1 + 2t + 2t^2 + 2t^3 + \ldots + 2t^{h-1}$$
$$= 1 + 2(t + t + t^2 + t^3 + \ldots + t^{h-1})$$
$$= 1 + 2\sum_{i=1}^{h} t^{i-1}$$

Depending on the height and degree of the B-tree, the nodes in the final B-tree are determined.

18.2-5

Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n.

Step 1 of 2

```
B-TREE-SEARCH(x,k)
1.  $i \leftarrow 1$
2.  while $i \le x.n$ and $k > x.key_i$
3.      do $i \leftarrow i+1$
4.  if $i \le x.n$ and $k = x.key_i$
5.     then return $(x, i)$
6.  if x. leaf
7.     then return NIL
8.  else DISK-READ $(x.c_i)$
9.     return B-TREE-SEARCH $(x.c_i, k)$
```

Step 2 of 2

The B-TREE-SEARCH within a node is implemented by binary search, then the time taken by the while loop of lines 2 and 3 in the search algorithm is $O(\lg t)$.

Then the total CPU time is $O(t \cdot h) = O(\lg t \cdot \log_t n)$

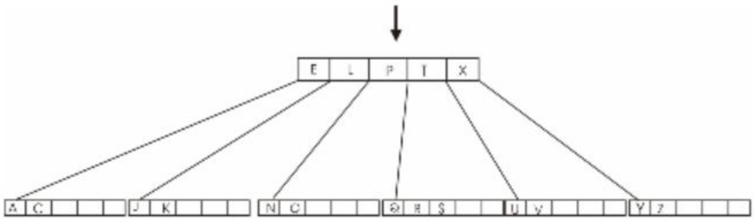
$$= O(\lg n)$$

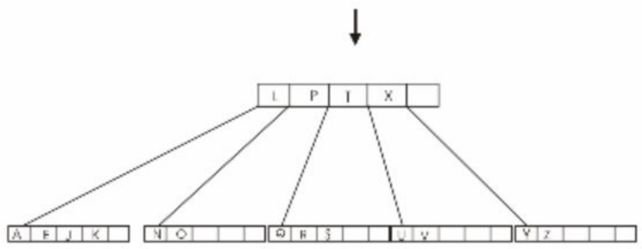
18.2-7
Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

18.3-1
Show the results of deleting C, P, and V, in order, from the tree of Figure 18.8(f).

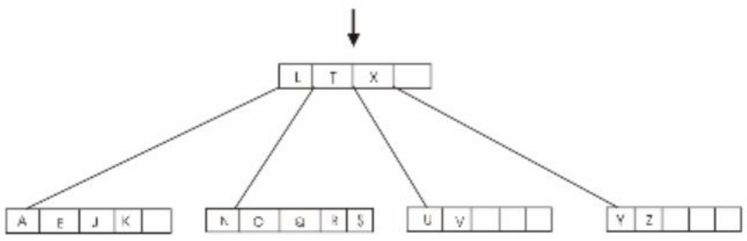
Step 1 of 2

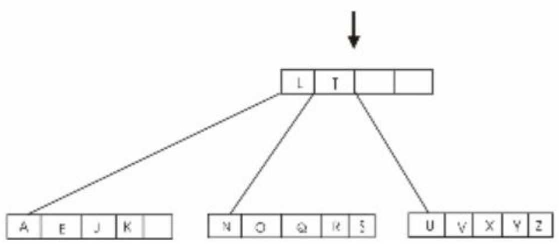
Fig:





Step 2 of 2





18.3-2
Write pseudocode for B-TREE-DELETE.

Step 1 of 1

Pseudo code for B-TREE-DELETE is:

```
B-TREE-DELETE-KEY(x, k)
1. if not x.leaf then
2. y = PRECEDING-CHILD(x)
3. z = SUCCESSOR-CHILD(x)
4. if y.n > t-1 then
5. $k'$ = FIND-PREDECESSOR-KEY (k, x)
6. MOVE-KEY ($k'$, y, x)
7. MOVE-KEY (k, x, z)
8. B-TREE-DELETE-KEY (k, z)
9. else if z.n > t-1 then
10. $k'$ = FIND-SUCCESSOR-KEY (k, x)
11. MOVE-KEY ($k'$, z, x)
12. MOVE-KEY (k, x, y)
13. B-TREE-DELETE-KEY (k, y)
14. else
15. MOVE-KEY (k, x, y)
16. MERGE-NODES(y, z)
17. B-TREE-DELETE-KEY (k, y)
18. else (leaf node)
19. y = PRECEDING-CHILD(x)
20. z = SUCCESSOR-CHILD(x)
21. w = root(x)
22. v = rootKey(x)
23.if x.n > t-1 then
24. REMOVE-KEY (k, x)
25. else if y.n > t-1 then
26. $k'$ = FINE-PREDECESSOR-KEY (w, v)
27. MOVE-KEY ($k'$, y, w)
28. $k'$ = FIND-SUCCESSOR-KEY (w, v)
29. MOVE-KEY ($k'$, w, x)
30. B-TREE-DELETE-KEY (k, x)
31. else if w.n > t-1 then
32. $k'$ = FIND-SUCCESSOR-KEY (w, v)
33. MOVE-KEY ($k'$, z, w)
34. $k'$ = FIND-PREDECESSOR-KEY (w, v)
```

```
35. MOVE-KEY ($k'$, w, x)
36. B-TREE-DELETE-KEY (k, x)
37. else
38. s = FIND-SIBLING (w)
39. $w'$ = w.root
40. If $w'.n$ =t-1 then
41. MERGE-NODES ($w'$, w)
42. MERGE-NODES (w,s)
43. B-TREE-DELETE-Key (k,x)
44. else
45. MOVE-KEY (v, w, x)
46. B-TREE-DELETE-KEY (k, x)
```

The key functions in this code are

1. Preceding-Child(x) Returns the left child of key x
2. Move-Key (k, n1, n2) Moves key k from node n1 to node n2
3. Merge-Nodes (n1, n2) Merges the keys of nodes n1 and n2 into a new node
4. Find-Predecessor-Key (n, k) Returns the key preceding key k in the child of node n
5. Remove-Key (k, n) Deletes key k from node n. n must be a leaf node

18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p, the top element is the $(p \bmod m)$th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.

To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)

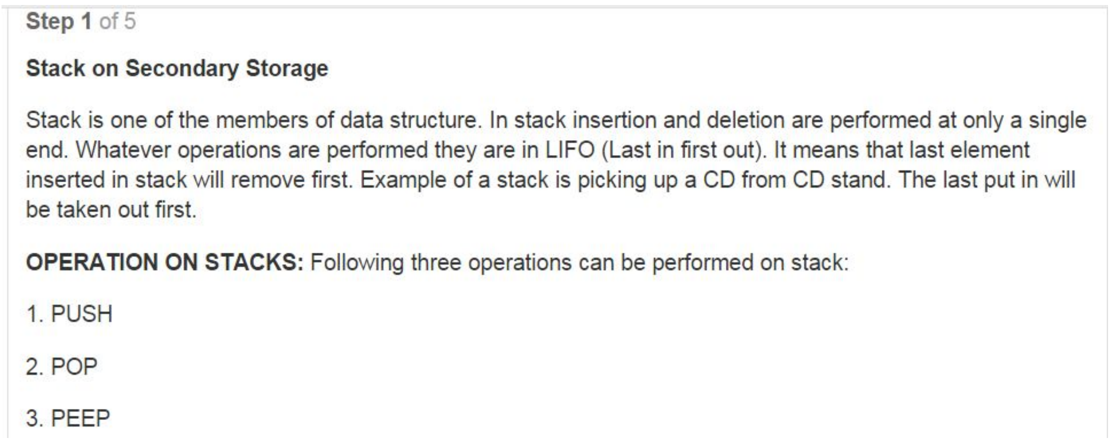
Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

c. What is the worst-case number of disk accesses required for n stack operations? What is the CPU time?

Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.



Step 1 of 5

Stack on Secondary Storage

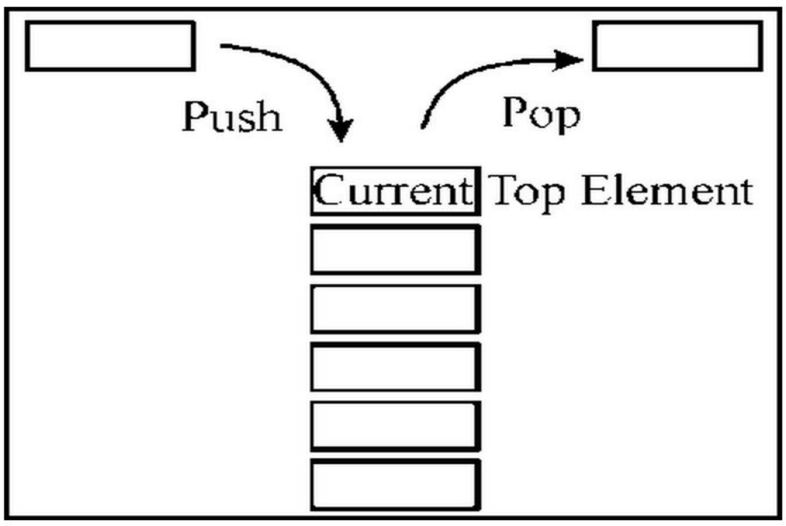
Stack is one of the members of data structure. In stack insertion and deletion are performed at only a single end. Whatever operations are performed they are in LIFO (Last in first out). It means that last element inserted in stack will remove first. Example of a stack is picking up a CD from CD stand. The last put in will be taken out first.

OPERATION ON STACKS: Following three operations can be performed on stack:

1. PUSH

2. POP

3. PEEP



1. **PUSH**: This is the operation performed for insertion. The push operation inserts a data item into the stack to its top.

2. **POP**: This is the operation performed for deletion. This pop operation works for removing a data item from the stack to its top.

3. **PEEP:** This operation is used only for reading the top element of the stack. It will not remove the top element of stack.

Step 2 of 5

a. Consider a stack implementation which holds one page of the stack in memory. The conditions states that stack operation only performed when enough disk pages present in memory. If the required disk pages are present in memory, then no disk accesses will be required. To find the worst case running time to perform n stack operation using the above strategy is:

Take the following alternating sequence:

PUSH, PUSH, POP, POP, $\cdots$

Assume that first PUSH operation done at the end of page. This is happening at the page boundary. The second PUSH operation in the above sequence requires writing the first most word of the next page. Similarly the second POP operation requires reading in the previous page once more. Accessing an *m* words page of disk requires $\Theta(m)$ CPU time and one disk access. Hence, n operation will require total $\Theta(n)$ disk accesses, as n stack operations will be performed.

Hence, the total CPU time taken will be $\boxed{\Theta(mn)}$.

b. If PUSH operations done on the pages which are not in the memory. Then each PUSH operation required to access page from disk and then store into memory. Therefore n operation will require total $\Theta(n)$ disk accesses, as n push stack operations will be performed. Accessing an *m* words page of disk requires $\Theta(m)$ CPU time and one disk access. Fully, a disk will be accessed one by one.

Hence, the CPU running time is $\boxed{\Theta(mn)}$.

Step 3 of 5

c. A stack consists of mainly two operations, PUSH and POP. The cost of implementation of any operation is calculated by the combination of total CPU time and number of disk access.

In worst case number of disk accesses required for *n* stack operation will be $\Theta(n)$. Whatever operation will perform on the stack, either it will be PUSH or POP. It will be implemented as a whole. And CPU running time implemented on this will be $\Theta(mn)$.

The reason behind is that any disk access to a page m words incurs charges of one disk access and $\Theta(m)$ CPU time and here number of disk accesses required for *n* stack operation will be $\boxed{\Theta(n)}$.

d. In the above parts it is mentioned that only one page of disk can reside in the memory. In order to implement any stack operation, it is necessary to keep the required current page in the memory.

Now, implement a stack by keeping two pages of disk in memory in order to keep the previously used pages in the memory. Now, uses two pages as discussed. Keep marking or tracking the pages to know which of the following two pages are in memory had been least recently used (LRU).

Whenever a new page is store into memory or can say that read a new page, save the LRU page to the disk if any change is made, and read it in the new page. While executing any PUSH operation, memory will be a_0 for a single page, further it would be a_0a_1 while performing on $(a+1)st$ page through $(2p)th$

pushes, further a_2a_1 and then a_2a_3.

Step 4 of 5

The stack pointer of disk will always point to the top of any fresh or new page to be read, taken from the disk. There is data and free space always available in page for performing stack operations in memory. Only two operations PUSH and POP can perform on stack before moving to the next page for reading. So, the stack operations should be performed before every disk access.

When one page resides in the memory then the cost of accessing an *m* words page of disk requires $\Theta(m)$ CPU time and one disk access. But now two pages of disk reside in the memory. So, that the cost of accessing the disk is reduces to $(1/m)th$ as cost of disk access, which is $O(1/m)$ for every stack operation performed. Because there will be more chance to perform Push or pop operation on currently two pages stored in memory. Here *m* is the number of words on disk access page.

Step 5 of 5

Now, two pages are store in the memory and each page has *m* words. So, there will be more probability to insert or get the word in the current pages. It will also reduce the CPU time by *m* because word is already in memory.

CPU time when one page in memory and each page has *m* words= $\Theta(m)$

CPU time when two pages in memory and each page has *m* words is

$$\Theta\left(\frac{m}{m}\right) = \Theta(1)$$

After 'a' stack operation have been performed, values of $O(a)$ are collected which could be paid for any required disk access.

Hence, the amortized CPU time will be $O(1)$ for any stack operation.

18-2 *Joining and splitting 2-3-4 trees*

The ***join*** operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.key < x.key < x''.key$. It returns a set $S = S' \cup \{x\} \cup S''$. The ***split*** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. In this problem, we investigate

how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.

a. Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.

b. Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k, the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'', respectively.

c. Consider the simple path p from the root of a 2-3-4 tree T to a given key k, the set S' of keys in T that are less than k, and the set S'' of keys in T that are greater than k. Show that p breaks S' into a set of trees $\{T'_0, T'_1, \ldots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \ldots, k'_m\}$, where, for $i = 1, 2, \ldots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i? Describe how p breaks S'' into sets of trees and keys.

d. Show how to implement the split operation on T. Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T''. The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T. (*Hint:* The costs for joining should telescope.)

Step 1 of 5

a) Every node of the tree, keep a field *x.height* is equal to zero if x is a leave, otherwise *x.height* is equal the length of a path from x to a leaf. With a few changes, the field is maintained within the procedure of merging. Therefore, the asymptotic running time in any operation (searching, insertion, and deletion) is the same.

Step 2 of 5

b) **Implement join operation:**

1) First takes h' > h", then insert k into the right most node N' at height h"+1 of T'. If N' is full, extract its largest key k'_1 fill k into its space and recursively insert k' into the parent of N'. Therefore the tree T" to the right of k.

2) Secondly take h' = h", create a new root with k, and attach T' and T" to the left and right of k.

3) And finally take h'<h" insert k into the leftmost node at height h'+1 of T".

The procedure is the similar to the first case since the procedure has at most |h'-h"|+1 recursions, each takes $O(1)$ time. The total time for joining is $0(|h'-h"|+1)$.

Step 3 of 5

c)
Suppose the path p has length l, then $P = \{x_1.c_{a_1},\ldots,x_1.c_{a_1},k\}$, where $a_i \in \{1,2,3,4\}$ for $i = 1,\ldots,l$ do:

a. if $c_i = 1$, then $k_i = \text{NIL}$ and $T_i^1 = \phi$

b. if $c_i = 2$, then $k_i = x.\text{key}_1$ and $T_i^1 = x_i.c_1$

c. if $c_i = 3$, then $k_i = x_i.\text{key}_2$ and $T_i^1 = A$

Tree formed by a root containing $x_i.\text{key}_1$ and two sub trees $x_i.c_1$ and $x_i.c_2$

d. if $c_i = 4$, then $k_i = x_i.\text{key}_3$ and $T_1' = A$

Tree formed by a root containing $x_i.\text{key}_1$ and $x_i.\text{key}_2$ and

three sub trees, $x_i.c_1, x_i.c_2$ and $x_i.c_3$.

Step 4 of 5

Finally consider the node x' containing k. If $k = x'.\text{key}_1$ then $T'_{i+1} = x'.c_1$. If $k = x'.\text{key}_2$, then T'_{i+1} = a tree formed by a root containing $x'.\text{key}_1$ and three sub trees $x_i.c_1, x_i.c_2$ and $x_i.c_3$. After tress $T_1,\ldots,T'_{l+1}$ and keys $k'_1,\ldots,k'_l$ are generated, remove all $T'_l = \phi$ and $k_i = \text{NIL}$. The result is a set of trees $\left\{T'_0,\ldots,T'_m\right\}$ and a set of keys $\left\{k'_1,\ldots k'_m\right\}$. By the properties of $2-3-4$ trees, these are the desired sets for $i = 1,\ldots,m$. The height of $T'_i\text{-}1$ is greater or equal or the height of T'_1. The procedure of break S''into sets of trees and keys is symmetric.

Step 5 of 5

d)

To split a tree, first find a path from the root to the keys $k = x.\text{key}$. Then break the set S' into sets of trees $\left\{T'_0,\ldots,T'_m\right\}$ and keys $k'_1,\ldots,k'_l$ as described above. Next for i from m to 1, do $\text{JOIN}\left(T'_i, k_i, T'_{i\text{-}1}\right)$. The result is a $2-3-4$ tree containing the set S'. Do the same for S''and create another tree for S''. Finally return S' and S'' to analyze the running time, observe that the time to join two trees of height h' and h'' is $O(1+|h'\text{-}h''|)$ by the (B). Then the total time to join all the trees in S' is:

$$\sum_{i=1}^{m}\left(1+\left|h\left(T'_{i\text{-}1}\right)\text{-}h\left(T'_i\right)\right|\right) = m + \sum_{i=1}^{m}\left[h\left(T'_{i\text{-}1}\right)\text{-}h\left(T'_i\right)\right]$$

$$= m + h\left(T'_0\right)\text{-}h\left(T'_m\right) \in O(\lg n)$$

since $m_1\ h\left(T'_0\right) \le h(T) \in O(\lg n), h\left(T'_m\right) \ge 0$

Since the time for finding the path breaking S' and S'' is also bounded by $O(\lg n)$. The running time for splitting is $O(\lg n)$.