CIS 5511. Programming Techniques

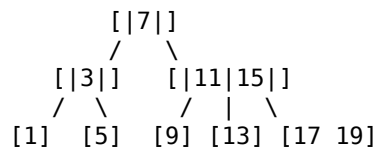
B-trees and Amortized Analysis

Various types of *binary* tree, including BST and heap, can be extended into trees where a node can have more than two successors. These data structures are similar to binary trees in certain aspects, but also have different properties. For example, Prefix tree (or trie) is an n-ary search tree where the key value is used one character (or digit) per level in the search process.

1. 2-3 trees and 2-3-4 trees

In a node of BST, 2 subtrees are separated by 1 key value (Left < key < Right), so the search scope is reduced. This idea can be extended to separate m subtrees using $m-1$ keys, and to keep the keys and the subtrees "sorted" as in BST. Such a tree can also be balanced in the sense that all the paths from the root to a leaf have the same length. This balance is achieved by allowing the number of keys in a node to vary in a range.

A 2-3 tree is such a tree where each node has 2 or 3 children (subtrees), and they are separated by 1 or 2 key values. Within the tree, a 2-node has 2 children and 1 key, while a 3-node has 3 children and 2 keys. All leaves are at the same level, and all keys are "sorted horizontally" as in a BST.



Searching a 2-3 tree is similar to searching a BST, except that within each 3-node, in the worst case two comparisons are needed.

Insertion: In a 2-3 tree, a new key value is initially inserted into a leaf where the search stops, according to the order of a search tree. Then there are the following possibilities:

1. If the leaf was a 2-node before the insertion, it becomes a 3-node, and the process stops.
2. If the leaf was a 3-node, it is split into two 2-node, and the middle value is inserted into the parent to separate the 2 nodes.
3. This process may repeat at the parent. If the root splits, a new root is generated and the height of the tree is increased by 1.

Deletion: To remove a key value from a 2-3 tree, the first step is to search for it. If it is in a leaf, simply remove it. If it is in an internal node, replace the deleted key by its successor (or predecessor), which must be in a leaf. Then there are the following possibilities:

1. If the node is not empty after losing a key, the process stops.
2. If the empty node has a 3-node sibling, the "nearest" key to the empty node is moved

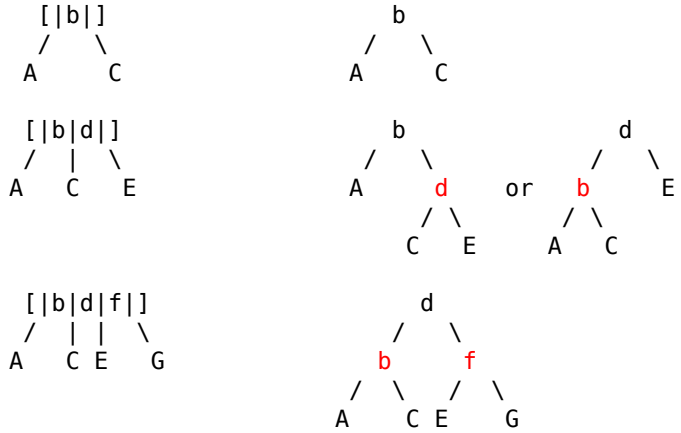
from the sibling to the parent, and the key in the parent separating the two nodes is moved into the empty leaf.
3. If the empty node has no 3-node sibling, it is merged with a 2-node sibling with the key from the parent separating them into a 3-node.
4. This process may repeat at the parent (if it becomes empty), where a subtree may be moved when a key from its sibling is moved. If the root becomes empty, the node is removed and its only child becomes the new root.

Since all leaves are at the same level, the complexity of major operations is O(log n).

2-3-4 trees extend 2-3 trees by allowing 4-nodes, so each non-leaf node can have 2, 3, or 4 children.

The operations of 2-3-4 trees are similar to those of 2-3 trees.

There is a mapping between a 2-3-4 tree and a Red-Black tree. In the following, a lower case letter represents a key, and an upper case letter represents a subtree.



Since each node in a 2-3-4 tree corresponds to one black node (plus at most two red nodes) in a Red-Black tree, the height of a 2-3-4 tree corresponds to the number of black nodes in the path from the root to a leaf in a Red-Black tree, which is half of the number of comparisons in the worst case.

2. B-Trees

B-trees are balanced search trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since a disk access takes much longer time than a memory access, it is desired to use larger nodes to reduce disk access, assuming that a node can be retrieved in each disk access.

A *B-tree with a minimum degree t* (*t* is 2 or more) has the following properties:

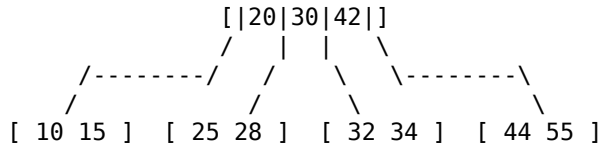
1. All leaf nodes are at the same level of the tree.
2. Every non-leaf node has at least *t* and at most *2t* children, with the exception that the root can have at least 2 children.
3. If a non-leaf node contains *n* keys, it contains *n + 1* references to its children.

4. All keys stored in a node are sorted.
5. All keys stored in a subtree are between two keys in its parent node, except the smallest and the largest ones.

So 2-3-4 tree is B-tree with degree 2.

Another convention is to define an *order-m* B-tree, where *m* is the maximum number of children of a node, and the minimum number cannot be less than half of *m* (i.e., is *ceiling(m/2)*) except the root can have as few as 2 children. Therefore 2-3 tree and 2-3-4 tree are B-tree of order 3 and 4, respectively. This definition is more general than the previous one using "degree", as the maximum number of children can be odd or even.

Example: an order-5 B-tree (where each node contains 2-4 keys and 3-5 children)

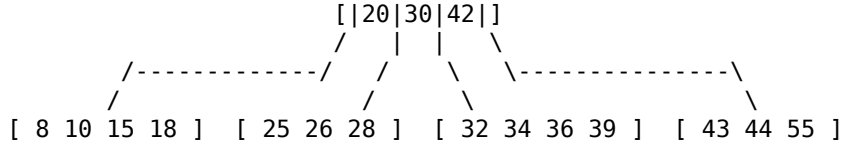


Search algorithm:

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

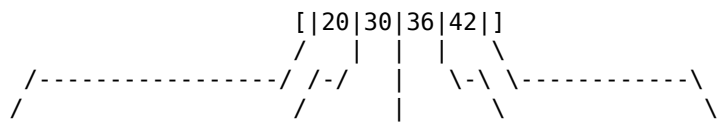
Insertion: Insert can be seen as a search (for the inserting position) followed by the actual insert of the key. As in 2-3 and 2-3-4 trees, insertion happens in a leaf node until the leaf node has *m* keys. Then it is split into two nodes and the middle key is sent up to the parent node.

Starting from the above B-tree, after adding 8, 18, 26, 36, 39, 43 we have



At this point two of the leaf nodes are full and two are not. Let's insert in one of the full leaf nodes and see what happens.

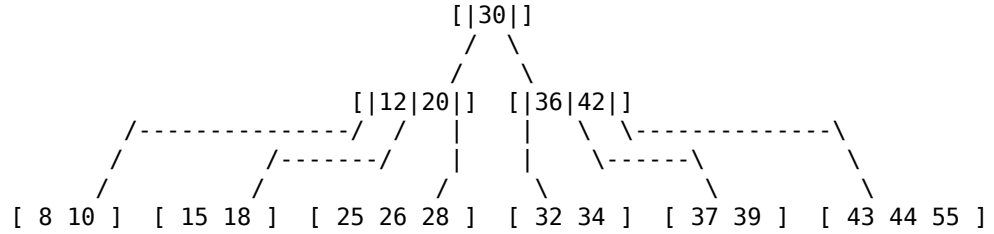
Insert 37: 37 is not in the tree, so it is inserted in the child node between 30 and 42. That node would contain 32 34 36 37 39, which is too big, so it splits into two nodes and pass the middle value (36) up.



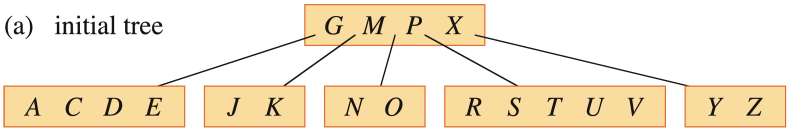
```
[ 8 10 15 18 ]  [ 25 26 28 ]  [ 32 34 ]  [ 37 39 ]  [ 43 44 55 ]
```

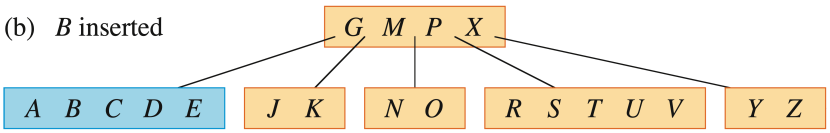
Now let's insert in the other node which is full. Remember that all insertions begin at a leaf node. Insert 12 in leftmost leaf: 8 10 12 15 18 -- too big, split and pass 12 up.

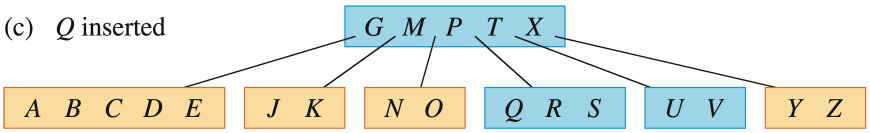
Parent node becomes 12 20 30 36 42 -- too big, split and pass 30 up to become the new root (it is fine if the root has less than *m/2* entries). From this example, we see that B-Trees actually grow in a bottom-up manner (at root), rather than top-down (at leaf).

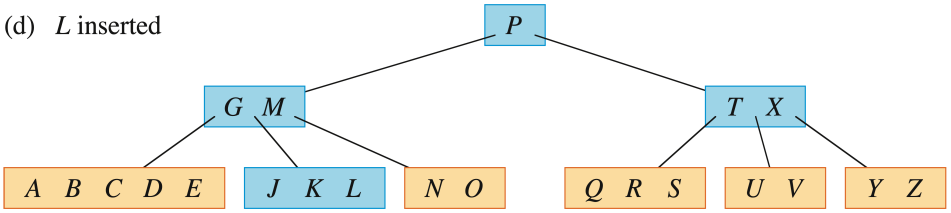


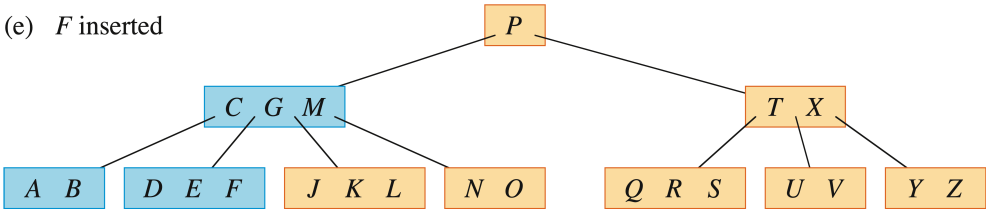
Textbook example (t = 3):











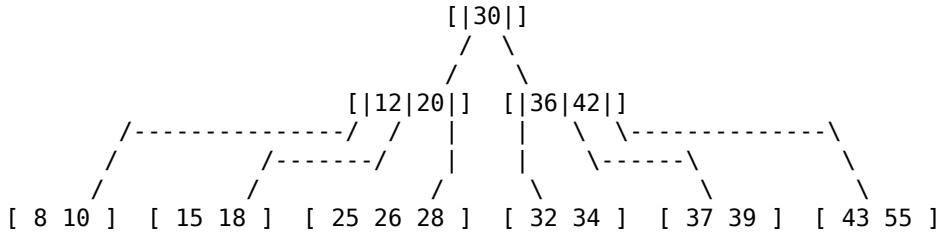
Deletion: For deletion, we can delete a key that is either in a leaf or a non-leaf node. Again, deletion consists of a search, and, if the key is found in the tree, an actual deletion of it.

If we delete a key in a leaf node, there is no problem unless the leaf node becomes too small. Then we have to merge it with a sibling leaf (plus the key in the parent separating them). If the resulting node has too many keys we have to split it in two and send the

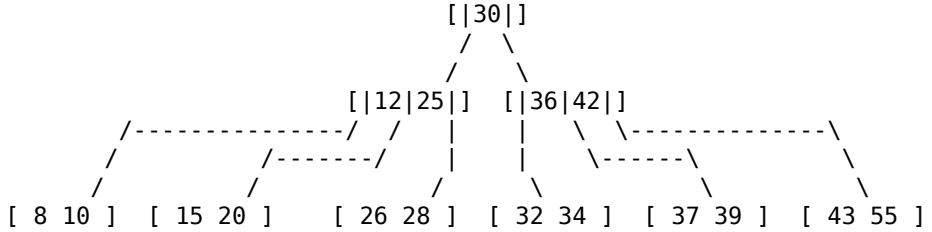
middle key up as in insertion.

If the key to be deleted is not in a leaf node, then it is replaced by the next larger key in the B-tree. Similar to finding the successor in a BST. Follow pointer to next child node and then follow all leftmost pointers until a leaf is reached. Replace key to be deleted with smallest key in the leaf node and then delete that key from the leaf. Merge with an adjacent leaf if necessary (see process for deleting from a leaf node).

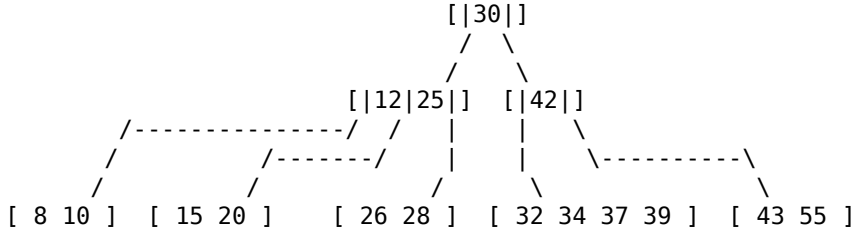
Let us delete 44 (in a leaf) from the above tree:



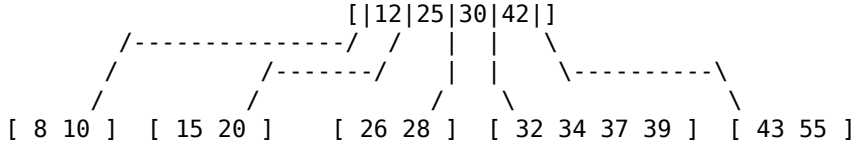
Now delete 18, in a leaf. Since the new leaf is too small, merge with the leaf to its right to become [15 20 25 26 28], then split and move 25 up. The total effect is like a rotation in an AVL Tree. It can be seen as getting a key from a sibling via the parent.



Delete 36, replace with 37, delete old 37, merge [39] with [32, 34], plus 37 -- finally get one full node.

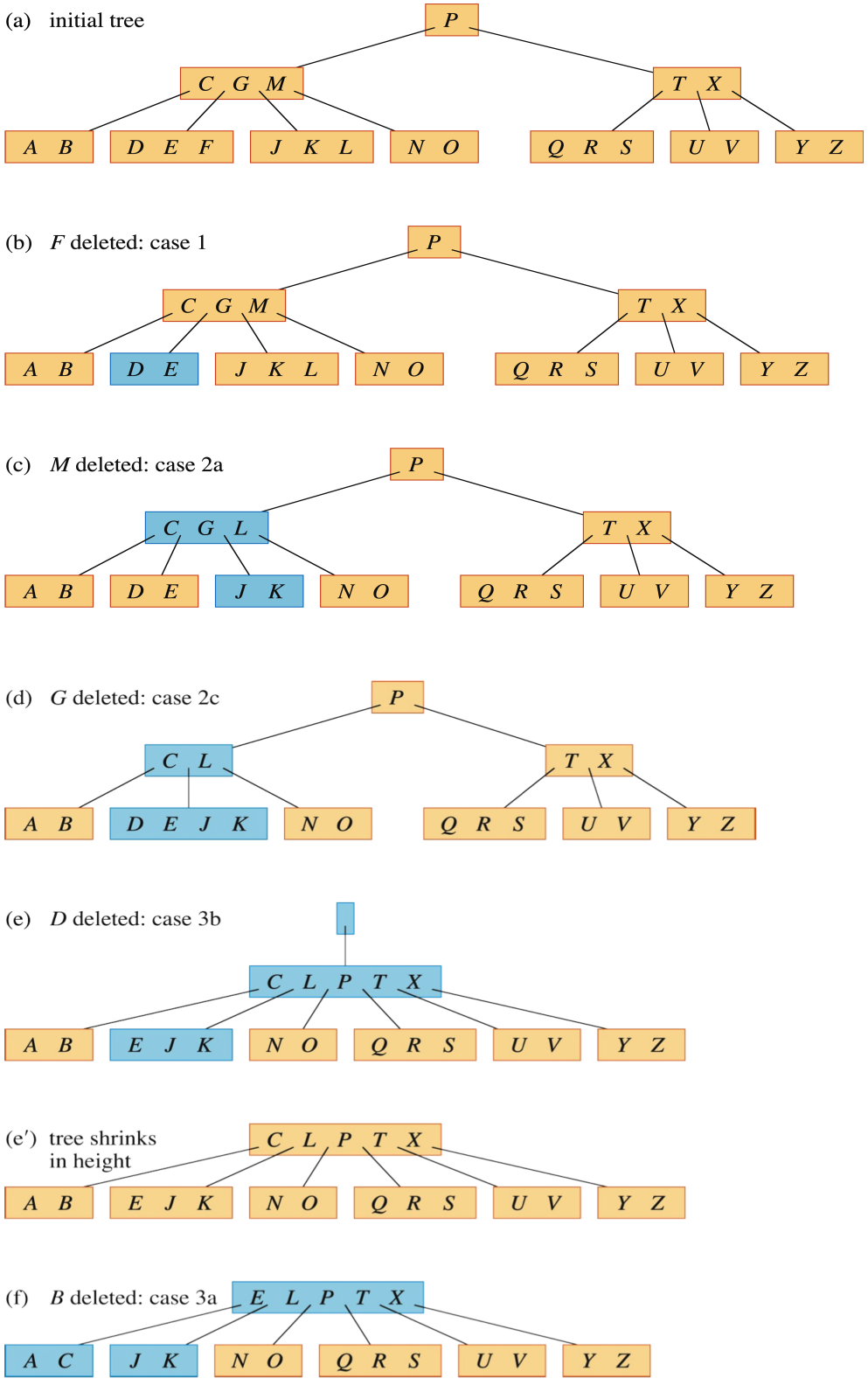


Now node with 42 has only one key, so it must be merged to become [12 25 30 42] which serves as the new root. Tree height is reduced by 1.



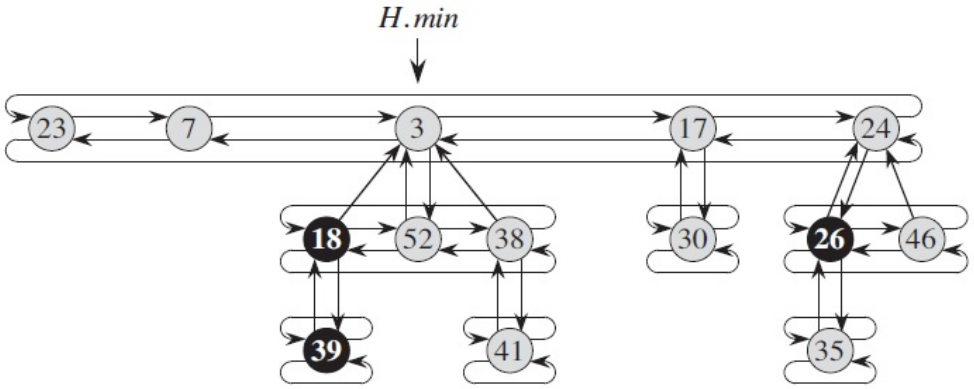
Every n-node B-tree has height O(lg n), and every major operation works on a path from the root to a leaf.

The delete algorithm in the textbook is slightly different, though with similar results. Example (t = 3):



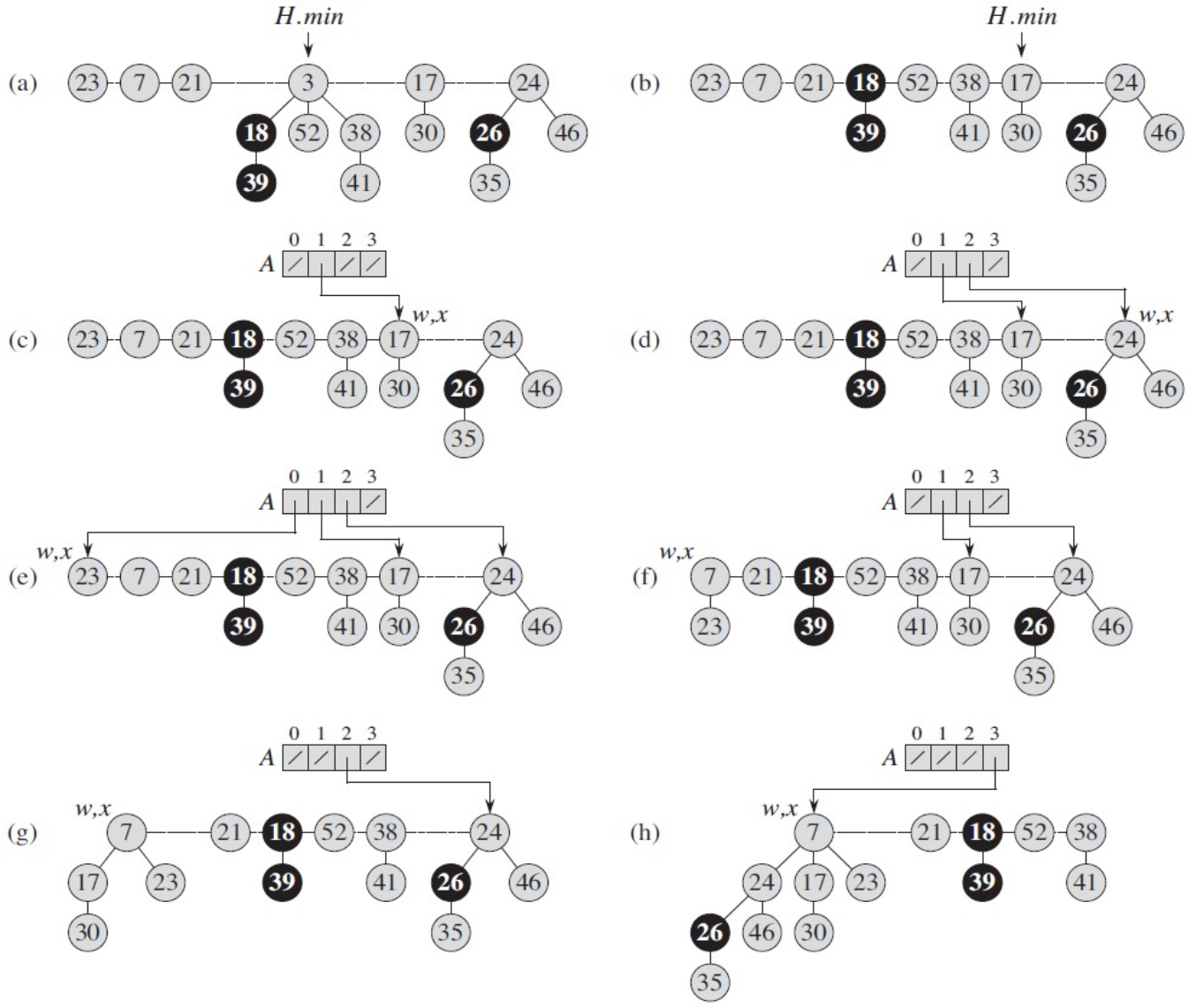
3. Fibonacci Heaps and Amortized Analysis

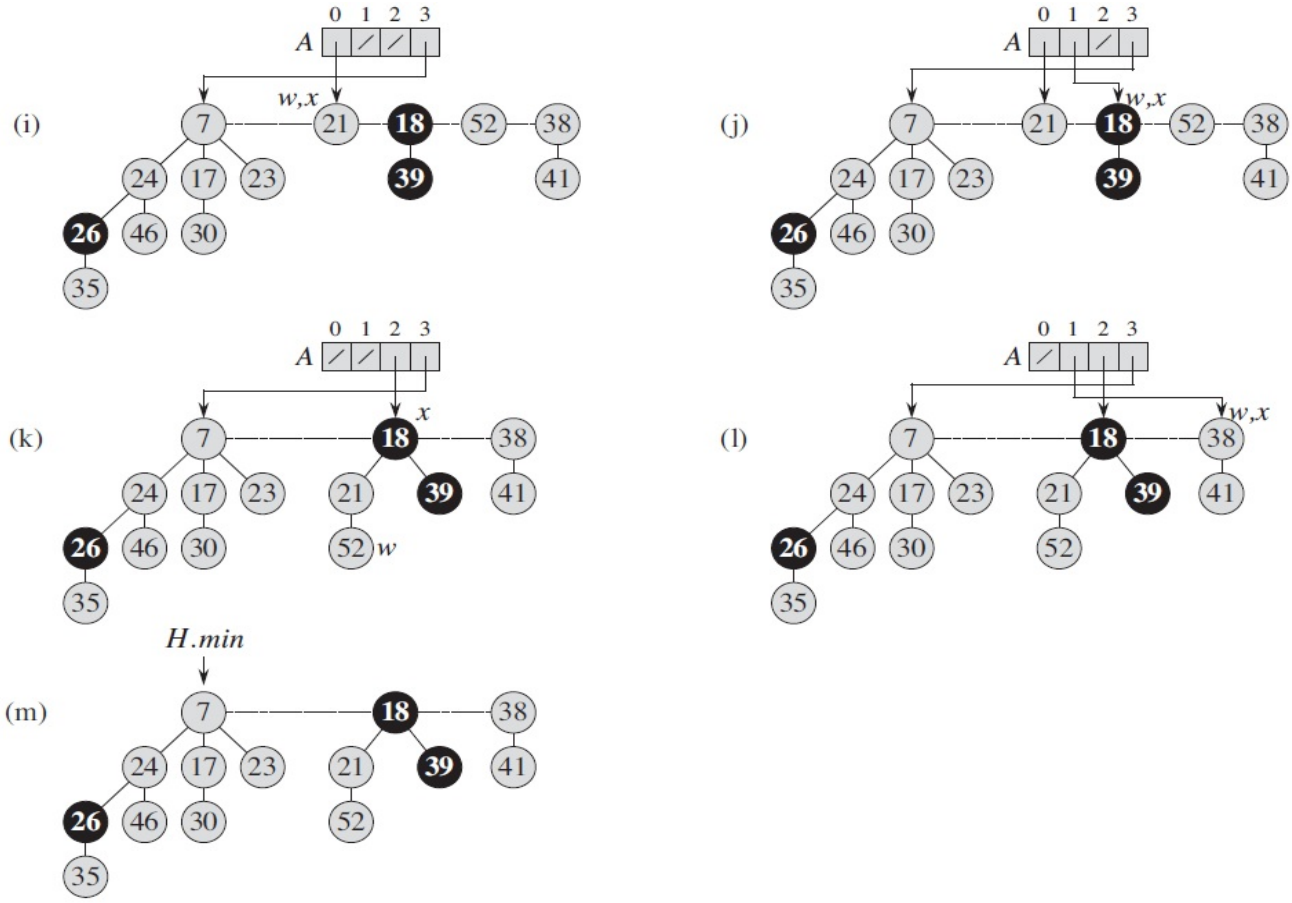
Similar to a (binary) heap, a Fibonacci heap is also often used to implement a priority queue. Such a heap is a collection of trees (not necessarily binary trees) with the min-heap property: the key of a node is smaller than the keys in its children. The root of the whole heap is the root of the tree that has the smallest key. Of course, it is possible to build a max-heap similarly.



In a Fibonacci heap, each node contains a pointer to its parent, a pointer to an arbitrary child, and two pointers to its siblings. The children of a node are linked together in a circular, doubly linked "child list" in an arbitrary order. The "degree" of a root indicates its number of children.

Defined in this way, certain operations can be carried out in O(1) time. INSERT simply adds the new node as the root of a new tree and update the heap root when the new key is smaller than the previous root; UNION combines two trees by let the larger root become a child of the smaller root. Complicated structure maintenance only happens after the minimum value (heap root) is removed. After that the children of the removed node are treated as roots of separate trees, then the trees of the same degree are union-ed repeatedly, as in the following example, where an array A is used to remember trees of each degree.


(a)
H.min
(b)
H.min
(c)
A
w,x
(d)
A
w,x
(e)
A
w,x
(f)
A
w,x
(g)
A
w,x
(h)
A
w,x



Compared to binary heap and self-balancing BST, Fibonacci heap and B-tree have more "relaxed" shape and order requirements, and some operations are executed in a "lazy" manner, i.e., postponing the work for a later time so several of them are combined into one. Consequently, some operations can take a long time while others are done very quickly.

For this type of structures, for each operation it usually makes more sense to analyze its average cost when repeated in the worst case, which is called the "amortized cost" operation. For example, if a stack is implemented as an array with fixed length n, operation *push* usually takes O(1) time, but the worst case is O(n) when it is full and space reallocation happens. Since the latter happens once after the former happens n times, the amortized cost of the operation is still O(1).

Here is a comparison between binary heap and Fibonacci heap:

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$