Algorithms

Study Notes

Zhijie Xia

Study Notes

github.com/zhijie-os
zhijiexia.website

Contents

Chapter 1

Dynamic Programming

1.1 Palindromic Subsequence

1.1.1 Longest Palindromic Subseqence

Problem Statment:

Given a sequence, find the length of its Longest Palindromic Subsequence(LPS). In a palindromic subsequence, elements read the same backward and forward.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Examples:

Input: "abdbca"
Output: 5
Explanation: LPS is "abdba"

Input: "cddpd"
Output: 3
Explanation: LPS is "ddd"

Basic Solution:

A basic brute-force solution could be to try all the subsequences of the given sequence. We can start processing from the beginning and the end of the sequence. So at any step, we have two options:

1. If str[begin]==str[end], increment counter by two. Subproblem: LPS in str[begin+1][end-1].
2. If str[begin]!=str[end], nothing. Subproblem: max LPS in str[begin+1][end] and LPS in str[begin][end-1].

Bottom Up Idea:

n = string.length ⇒ dp[n][n] and the subproblem is $dp[i][j]$: the LPS in substring str[i:j].
The solution to dp[i][j]:

1. str[i] == str[j] ⇒ dp[i+1][j-1]+2
2. str[i] != str[j] ⇒ max(dp[i+1][j],dp[i][j-1])

The solution to the LPS original problem is dp[0][n] which is the top right corner. Also, in 2×2 block, top right corner is based on the surrounding three.
⇒ populate the table in the following order:

1. From left to right.
2. From bottom to top.

Bottom Up Code:

```
class LPS{
    public:
        int findLPSLength(const string &st){
            vector<vector<int>> dp(st.length(), vector<int>(st.length(), 0));

            // every sequence with one element is a palindrome of length 1
            for (int i = 0; i < st.length(); i++) {
                dp[i][i] = 1;
            }

            // rows from bottom to up
            for (int startIndex = st.length() - 1; startIndex >= 0; startIndex--) {

                // column from left to right
                for (int endIndex = startIndex + 1; endIndex < st.length(); endIndex++) {

                    // case 1: elements at the beginning and the end are the same
                    if (st[startIndex] == st[endIndex]) {

                        dp[startIndex][endIndex] = 2 + dp[startIndex + 1][endIndex - 1];
                    }
                    else { // case 2: skip one element either from the beginning or the end

                        dp[startIndex][endIndex] =
                            max(dp[startIndex + 1][endIndex], dp[startIndex][endIndex - 1]);
                    }
                }
            }

            return dp[0][st.length() - 1];
        }
    }
```

1.1.2 Longest Palindromic Substring:

Problem Statment

Given a string, find the length of its Longest Palindromic Substring (LPS). In a palindromic string, elements read the same backward and forward.

Examples:

Input: "abdbca"
Output: 3
Explanation: LPS is "bdb"

Input: "cddpd"
Output: 3
Explanation: LPS is "dpd"

Chapter 2

Advanced Design and Analysis

2.1 Greedy Algorithm

A greedy algorithm always makes the choice that looks best at the moment. That is , it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

2.1.1 An activity-selection problem

Suppose we have a set $S = \{a_1, a_2, ..., a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.

Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i \leq f_i \leq$ inf. a_i and a_j are compatible if $s_i > f_j$ or $s_j > f_i$.

In the activity-selection problem, we wish to select a maximum-size subset of mutually compatible activities.

Assume that the activities are sorted in increasing order of finish time

$$f_1 \leq f_2 \leq f_3 ... f_{n-1} \leq f_n$$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

The approach is try to solve with dynamic programming and observe that only one choice should be consider - the greedy choice, that is DP with one subproblem.

Let c[i,j] denoted the maximum size of an optimal solution for set S_{ij}. $S_{ij} = \{a_k : f_i \leq s_k \leq f_k \leq s_j\}$.

The reccurence of choosing a_k

$$c[i, j] = c[i, k] + c[k, j] + 1$$

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Making the greedy choice

Intuition suggests that we should choose an activity that leaves the resource available for as many other activities as possible: choose the earliest finish time.

By choosing "the" activity, we only left one subproblem.

Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after a_k finishes.

But why is the intuition correct?

Theorem 16.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ∎

Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Assume that the n input activities are already ordered by monotonically increasing finish time.

In order to start, we add the fictitious activity a0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S.

```
RECURSIVE–ACTIVITY–SELECTOR(s,f,k,n)
{
    // select the earliest finish time activity
    int m = k+1;
    // such that s_m >= f_k (k is the last choice)
    while( m <= n && s[m] < f[k])
    {
        m = m+1;
    }

    if(m <= n)
    {
        // subproblem, maximum mutable compatible set of S_m
        return union(a_m,RECURSIVE–ACTIVITY–SELECTOR(s,f,m,n));
    }

    return []
}
```

Time Complexity: O(n) which is very obvious.

An iterative greedy algorithm

```
GREEDY–ACTIVITY–SELECTOR(s,f)
{
    int n = s.length;
    A = [a1];
    int k = 1;
    for(int m=1; m<n; m++)
    {
        if s[m] > f[k]
        {
            A.append(a_m);
            k = m;
        }
    }

    return A;
}
```

2.1.2 Elements of the greedy strategy

The greedy strategy doesn't always yield an optimal solution.

In order to use greedy method:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to iterative algorithm.

In the activity-selection problem, we proved that a greedy choice (the earliest finish time a_m to finish in S_k), combined with an optimal solution to the reaming S_m of compatible activities, yields an optimal solution to S_k.

More generally,

1. Think the optimization is to make a choice and solve the remaining subproblem.
2. Prove that there is always an optimal solution to the original problem that makes greedy choice $\Leftrightarrow$ the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice, we arrive at an optimal solution to the original problem.

Greedy-choice property

A dynamic programming algorithm proceeds bottom up, whereas a greedy strategy usually process in a top-down fashion. Of course, we must prove that a greedy choice at each step yields a globally optimal solution.

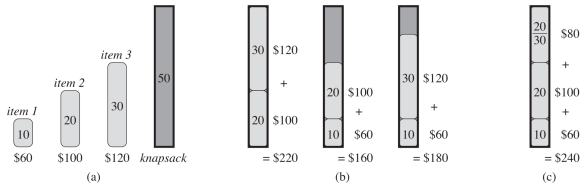
Optimal substructure

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

Dynamic Programming vs Greedy Algorithm

0/1 knapsack problem vs fractional knapsack problem.

0/1 knapsack is actually a dynamic programming problem while fractional knapsack can be solved greedily.



fractional knapsack can be solved by computing the value per pound and ordering the value per pound in increasing order, and then apply greedy strategy to attain the global optimal.

However 0/1 knapsack cannot be solved by the above strategy, since picking item 10 make introduce unusable space left, therefore, lowering the actual value per pounds.

2.1.3 Huffman Codes

Suppose there is 100,000-character data file that we wish to store compactly. We count the frequencey of each word.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Here, we use **binary character code** in which each character is represented by a unique binary string, which we can call **codeword**.

By considering the frequency of each character, a **variable-length code** can do better than a fixed-length code which requires 300,000 bits.

From above table, the total number of bits that needed

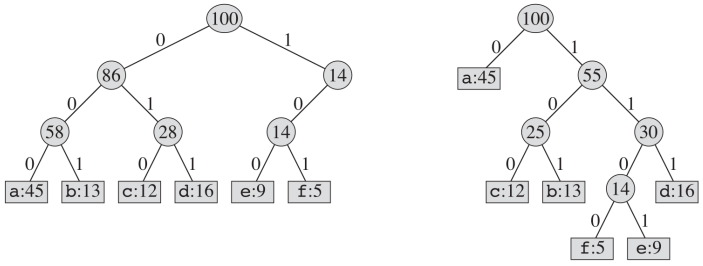
$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$$

Prefix codes

Note, in the above codewords. No codeword is a prefix of some other codeword.

Prefix codes are desirable for decoding, chop each codeword then deal with the remaining. It is unambiguous. For example, 00101101 would be 0-0-101-1101, that is **aabe**.

We use binary tree to represent for prefix code, so that we can easily pick off the initial codeword: "descend left" is 0, "descend right" is 1. An optimal code for a file is always represented by a full binary tree that is each nonleaf node has two children.



(a) (b)

If C is the alphabet, then the optimal prefix tree has —C— leaves. 1 leaf for each letter, and exacly —C—-1 internal nodes. Note that the value of the trees, the tree is branched by the frequency of letters.

Given an optimal prefix tree, we can easily compute the nubmer of required to encode a file. For each c in C, let c.freq denote the frequency of c and $d_T(c)$ denote the depth of c's leaf in the tree. $d_T(c)$ is also the number of bits for the codeword of c.

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a **Huffman code**.

Assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute c.freq giving its frequency.

```
HUFFMAN(C)
{
    int n = |C|;

    /* Q is a min-priority queue, such that the key the frequency
       also, for each c in the C is already a node.
    */
    Q = C;

    for(int i=0; i<n-1;i++)
    {
        new node z;

        // extract the least two frequent node (letters)
        // the order is arbitrary
        x = EXTRACT-MIN(Q);
        y = EXTRACT-MIN(Q);
        z.left = x;
        z.right = y;
```

```
            // this is like merging two letters x,y into one "letter" z.
            z.freq = x.freq + y.freq;
            INSERT(Q,z);
        }

        // return the root
        return EXTRACT_MIN(Q);
    }
```

Time complexity: since Q is a priority queue, the EXTRACT-MIN() takes O(lgn), and the for loops execute exactly n-1 times. That is O(nlgn). By replacing the priority queue with van Emde Boas tree, we can further reduce the T(n) to O(nlglgn).

Correctness of Huffman's algorithm

To prove that the greedy algorithm H UFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal- substructure properties.

Chapter 3

Advanced Data Structures

3.1 B-Trees

3.1.1 Introduction

B-tree are self balanced search trees designed to work well on disks.

3.1.2 Definition of B-trees

1. Every node x has the following attributes:
   (a) x.n: the number of keys currently stored in node x.
   (b) x.n keys: $x.key_1 \leq x.key_2 ... \leq x.key_n$
   (c) x.leaf: a boolean value that indicates whether the node is a leaf of internal node.
2. Internal node has x.n+1 pointers to its children: $x.c_1, x.c_2, ...x.c_{n+1}$
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the substree with root $x.c_i$, then
$$k_1 \leq x.key_1 \leq k_2 \leq ... \leq x.key_{x.n} \leq k_{x.n+1} \tag{3.1}$$
4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree
   (a) Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least 1 key.
   (b) Every node may contain at most 2t-1 keys. Therefore, an internal node may have at most 2t children. We say that a node is full if it contains exactly 2t-1 keys.

Simplest B-tree occurs when t=2, since t=2. The node can have 1-3 keys within a single node. Therefore, every internal node can have 2, 3, or 4 children. Thus it is call 2-3-4 Tree.

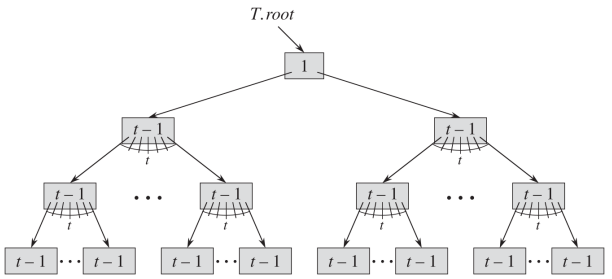
The height of a B-tree

The number of disk accesses required for most operations on B-tree is proportional to the height of the B-Tree.

Theorem:

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq log_t \frac{n+1}{2}$$



$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &= 2t^h - 1 \end{aligned}$$

3.1.3 Create, Search and Insert

Two convention:

1. The root is always in the main memory; no DISK-READ(root), but can have DISK-WRITE(root).
2. All nodes passed as parameters must already have been DISK-READ.

Search

```
B-TREE-SEARCH(x,k){
    i = 1;

    // find the smallest index i such that k <= x.key_i
    while(i <= n && k > x.key_i){
        i = i + 1;
    }

    // key is found in this level
    if(i<=x.n && k==x.key_i){
        return (x,i);
    }

    //reached to the leaf, and still not found
    else if(x.leaf){
        return NIL;
    }

    else{
        // bring the next node into the main memory
        DISK-READ(x.c_i)
        // search in the sandwitched child
        return B-TREE-SEARCH(x.c_i,k);
    }
}
```

Time Complexity:

Within each node, there is a linear search that would takes up to 2t-1 operations. There would be potiential $O(h) = O(log_t n)$ calls to reach the leaf. Thus a total CPU time:$O(th) = O(tlog_t n)$.

Although, we perform $O(h) = O(log_t n)$ disk operations.

Create

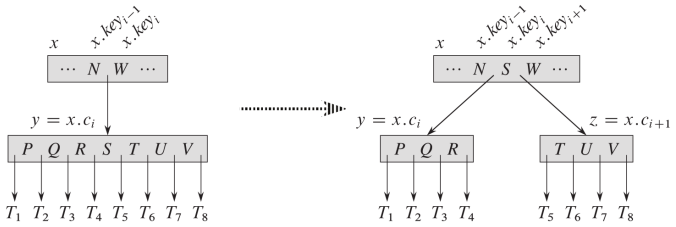
To create a B-tree, we need to allocate the root in the memory, and write the root into the disk.

```
B–TREE–CREATE(T){
    x = ALLOCATE–NODE()
    x.leaf = True
    x.n = 0
    DISK–WRITE(X)
    T.root = x
}
```

Time Complexity: The total would be O(1) disk operation, O(1) CPU usage.

Split

Assume, x is a nonfull node and its child $x.c_i$ is full node that both in the main memory. We could split the $x.c_i$ by the median and insert the median into x.




```
// x is the parent, i is the place of the spliting child
B–TREE–SPLIT–CHILD(x,i){
    // allocate the space for the resulting new half
    z = ALLOCATE–NODE();
    y = x.c[i];

    // clone the leaf property for new half
    z.leaf = y.leaf;
    z.n = t-1;

    // both y and z would have t-1 keys
    // take away y's keys
    for(j=1 to t-1){
        z.key[j] = y.key[j+t]
    }

    // take away the y's children
    if(!y.leaf){
        for(j=1 to t){
            z.c[j] = y.c[j+t]
        }
    }

    // set the number of keys in y
    y.n = t-1;

    // shift children one place right
    for(j=x.n+1 downto i+1){
        x.c[j+1] = x.c[j]
    }

    // attach the new half to the parent
    x.c[i+1] = z
```

```
        // shift the keys a position right
        for(j=x.n downto i){
            x.key[j+1] = x.key[j]
        }

        // insert the median into the parent
        x.key[i] = y.key[t]
        x.n = x.n+1

        // write back to the disk to make it effect
        DISK-WRITE(y)
        DISK-WRITE(z)
        DISK-WRITE(x)

    }
```

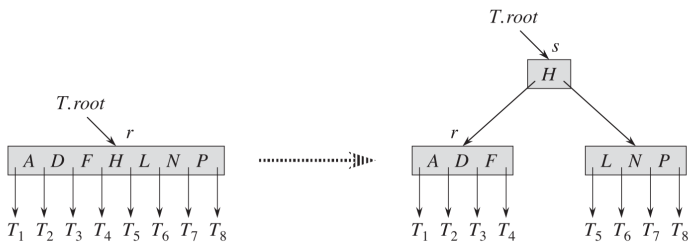
Time Complexity:
The CPU time would be O(t). And O(1) disk operations.

Insert

It would be smart to split along the searching insertion place and insert in one pass.

```
// T would be the B-tree, and k would be the key
B-TREE-INSERT(T,k){
    r = T.root
    // check whether the root is full
    if(r.n == 2t-1){
        // if it is, create a new root and allocate in the memory

        s = ALLOCATE-NODE()
        T.root = s
        s.leaf = FALSE
        s.n = 0

        // s is empty, it doesn't matter whether to attach the child
        s.c[1] = r

        // the split would add the median of orignal root into the new root
        B-TREE-SPLIT-CHILD(s,1)

        // Non root case insert
        B-TREE-INSERT-NONFULL(s,k)
    else{
        // Non root case insert
        B-TREE-INSERT-NONFULL(r,k)
    }

    }
}
```

The if part would look like



Note: splitting is the only way to increase the height.

After spliting the root, the B-TREE-INSERT-NONFULL would inserts key k. The precondition of B-TREE-INSERT-NONFULL is that the given node is not full.

```
// x is current node, and k is the key
B-TREE-INSERT-NONFULL(x,k){
    i = x.n

    // leaf is the base case
    if(x.leaf){

        // shift nodes,which are greater than k, one place right
        while(i>=1 && k<x.key[i]){
            x.key[i+1] = x.key[i]

            // also, found the insertion position
            i = i-1
        }

        // insert the key
        x.key[i+1] = k
        x.n = x.n+1

        // write back to the disk
        DISK-WRITE(x)
    }
    else{
        // found the child to be inserted into
        while(i>=1 && k<x.key[i]){
            i = i-1
        }
        i = i+1

        // read the child into memory
        DISK-READ(x.c[i])

        // if the child is full, split
        if(x.c[i].n == 2t-1){
            B-TREE-SPLIT-CHILD(x,i)

            // if the key is greater than the promoted median
            if(k > x.key[i]){
                i = i+1
            }
        }

        // recurse one level down
        B-TREE-INSERT-NONFULL(x.c[i],k)
    }
}
```

Time Complexity
To descend to a leaf, O(h) disk operations. And the total CPU time used is $O(th) = O(tlog_t n)$.

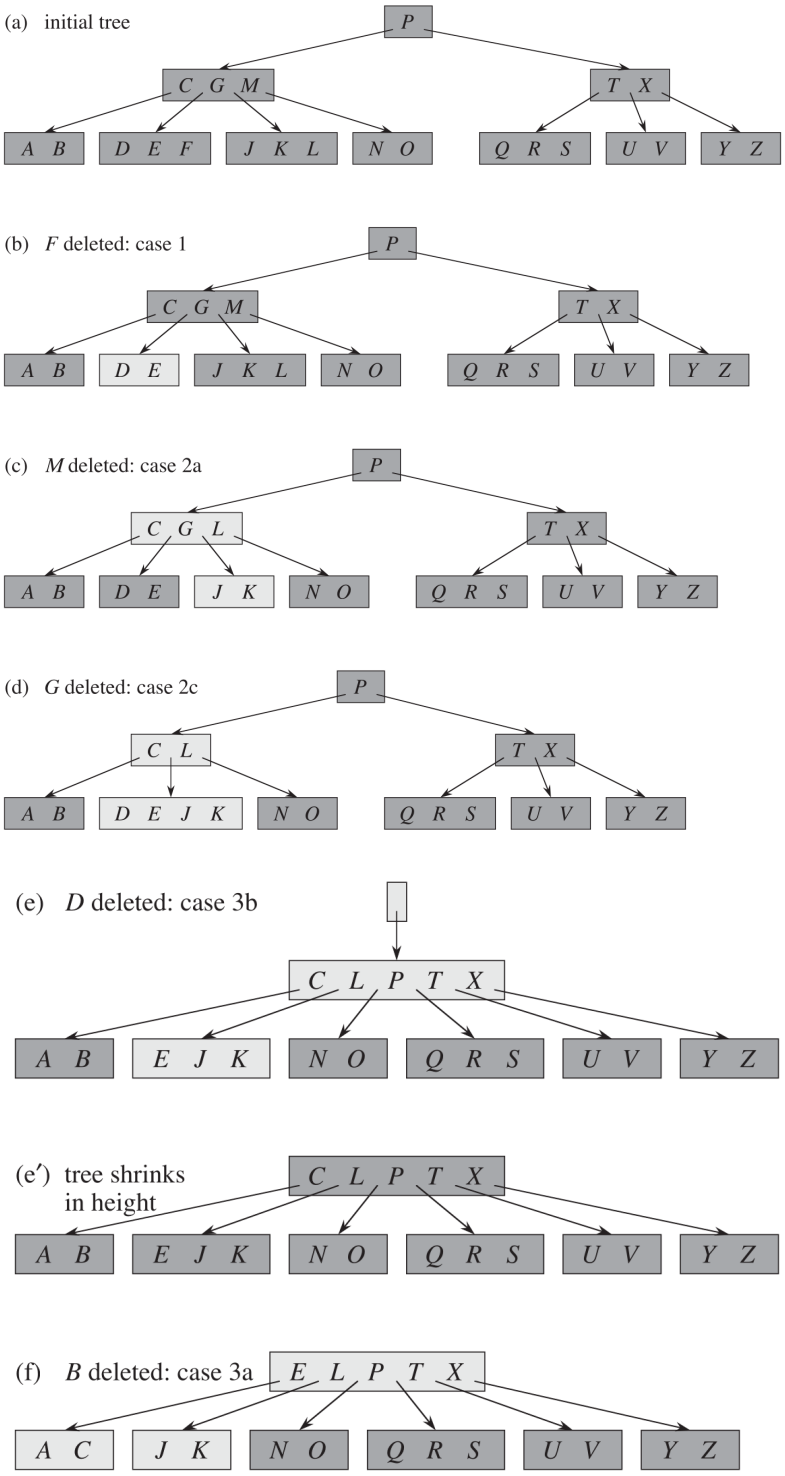
3.1.4 Deletion

The deletion is more complicated because

1. Deletion can happen in any node, not just leaf.
2. A node(with t-1 keys) can be underflowed due to deletion.

B-TREE-DELETE deletes the k from the subtree rooted at x. And whether the procedure call itself recursively on node x, it is guaranteed that x has at least t keys.

Case study



1. If the key k is in node x and x is a leaf, delete the key from x
2. If the key k is in node x and x is an internal node, do following
   (a) If the child y that precedes k in the node x has at least t keys, then find the predecessor k' of k in the substree rooted at y. Recursively delete k', and replace k by k' in y.
   (b) If y has fewer than t keys, then symmetrically, examine the child z that follows k in the node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x.
   (c) Otherwise, if both y and z have only t-1 keys, merge k and all of z into y, so that x loses both

k and the pointer to z, and y now contains 2t-1 keys. Then free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $x.c[i]$ of subtree that contains k, if k exist. If $x.c[i]$ has only t-1 keys, execute 3a or 3b to make sure to descend to a node contains at least t keys. Then recursively delete k from the subree that contains k.
   (a) If $x.c[i]$ has only t-1 keys but has an immediate sibling with at least t keys, give $x.c[i]$ an extra key by moving a key from x down to $x.c[i]$, moving a key from $x.c[i]$'s immediate sibling up to x, and moving the appropriate child pointer from the sibling into $x.c[i]$.
   (b) If $x.c[i]$ and both of $x.c[i]$'s immediate siblings have t-1 keys, merge $x.c[i]$ with one sibling, which invokes moving a key from x down into the new merged node to become the median key for that node.

Time Complexity:
O(h) disk operations. CPU time required is $O(th) = O(tlog_t n)$.
B-TREE-DELETE is the only way to decrease the height.

3.2 Fibonacci Heap

Fibonacci heap serves a dual purpose

1. supports a set of operations that constitutes "mergeable heap".
2. several fibonacci-heap operations run in constant amortized time.

Fibonacci Heap is favored when the number of EXTRACT-MIN() and DELETE() is relative small to the total number of operations. Some graphic problems have a favor on Fibonacci Heap.

From a practical point of view, however, the constant factors and program- ming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications, except for certain applications that manage large amounts of data.

3.2.1 Supported operations and time complexity

Mergeable heap supports:

1. MAKE-HEAP(): create and return a new heap containing no elements
2. INSERT(H,x): insert element x
3. MINIMUM(H): return the pointer to the minium element in heap H
4. EXTRACT-MIN(H):deletes the element from heap H whose key is minimum, re- turning a pointer to the element.
5. UNION(H1,H2):creates and returns a new heap that contains all the elements of heaps H1 and H2 . Heaps H1 and H2 are "destroyed" by this operation.

In addition to above operations, the Fibonacci Heap also supports:

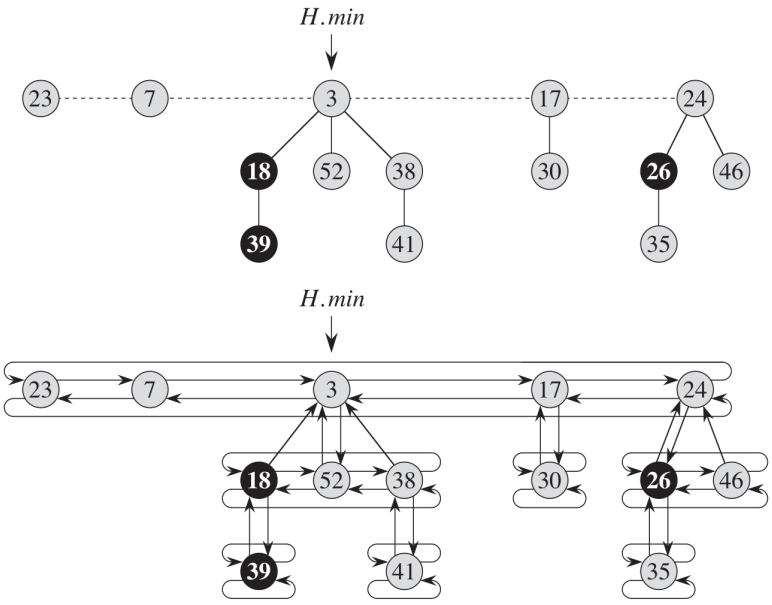
1. DECREASE-KEY(H,x,k):assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.
2. DELETE(H,x):deletes element x from heap H .

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	$\Theta(\lg n)$	$O(\lg n)$

Note the running time of Fibonacci Heap operations are amortized.

3.2.2 Structure of Fibonacci Heaps

1. A Fibonacci Heap is a collection of rooted trees that are **min-heap ordered**.
2. Each node x contains a pointer to x.parent and a pointer x.child to any of its child.
3. The children of node x are doubly-linked together in a circular. The list is called **child list** of x.
4. Each node x has x.degree what is the number of children in x's children list. x has x.mark indicates whether node x has lost a child since it has become a child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node
5. There is H.min that points to the minimum which is also a root of tree.
6. The roots of trees are also circular doubly-linked together, called **root list**.
7. H.n : the number of nodes inside the tree.



In the doubly-circular-linked list, one can insert or remove a node from any location in O(1) time. Also, given any two CDLLs, we can concatenate them in O(1) time.

Potential Function

t(H): the number of trees in the root list of H.
m(H): the number of marked nodes in H.
Potential Function

$$\phi(H) = t(H) + 2m(H)$$