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 Data Structures

2.1 B-Trees

2.1.1 Introduction

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

2.1.2 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

$$k_1 \leq x.key_1 \leq k_2 \leq ... \leq x.key_{x.n} \leq k_{x.n+1} \tag{2.1}$$

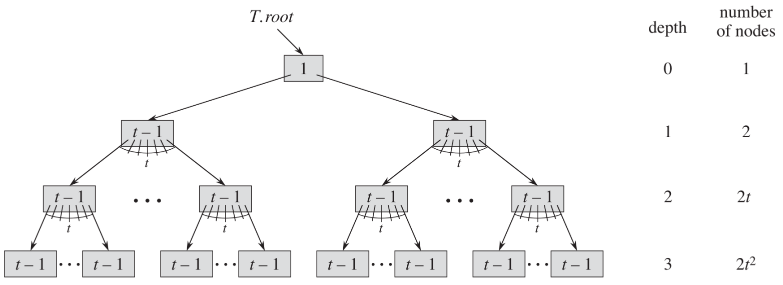
4. All leaves have the same depth, which is the tree's height h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the minimum degree of the B-tree
   (a) Every node other than the root must have at least 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 $t \geq 2$,

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



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

2.1.3 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

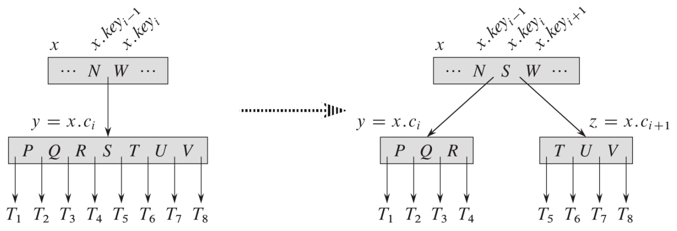
To create a B-tree, we need to allocate the root in the memory, and write the root into the disk.

```
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)

    }
```

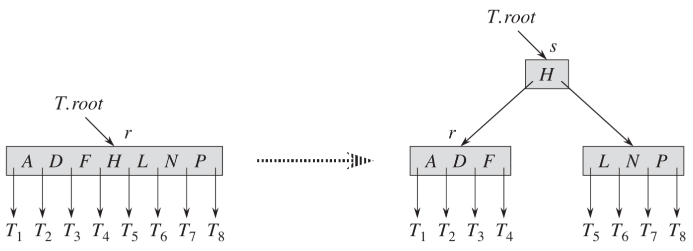
Time Complexity:
The CPU time would be O(t). And O(1) disk operations.

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 $O(th) = O(tlog_t n)$.

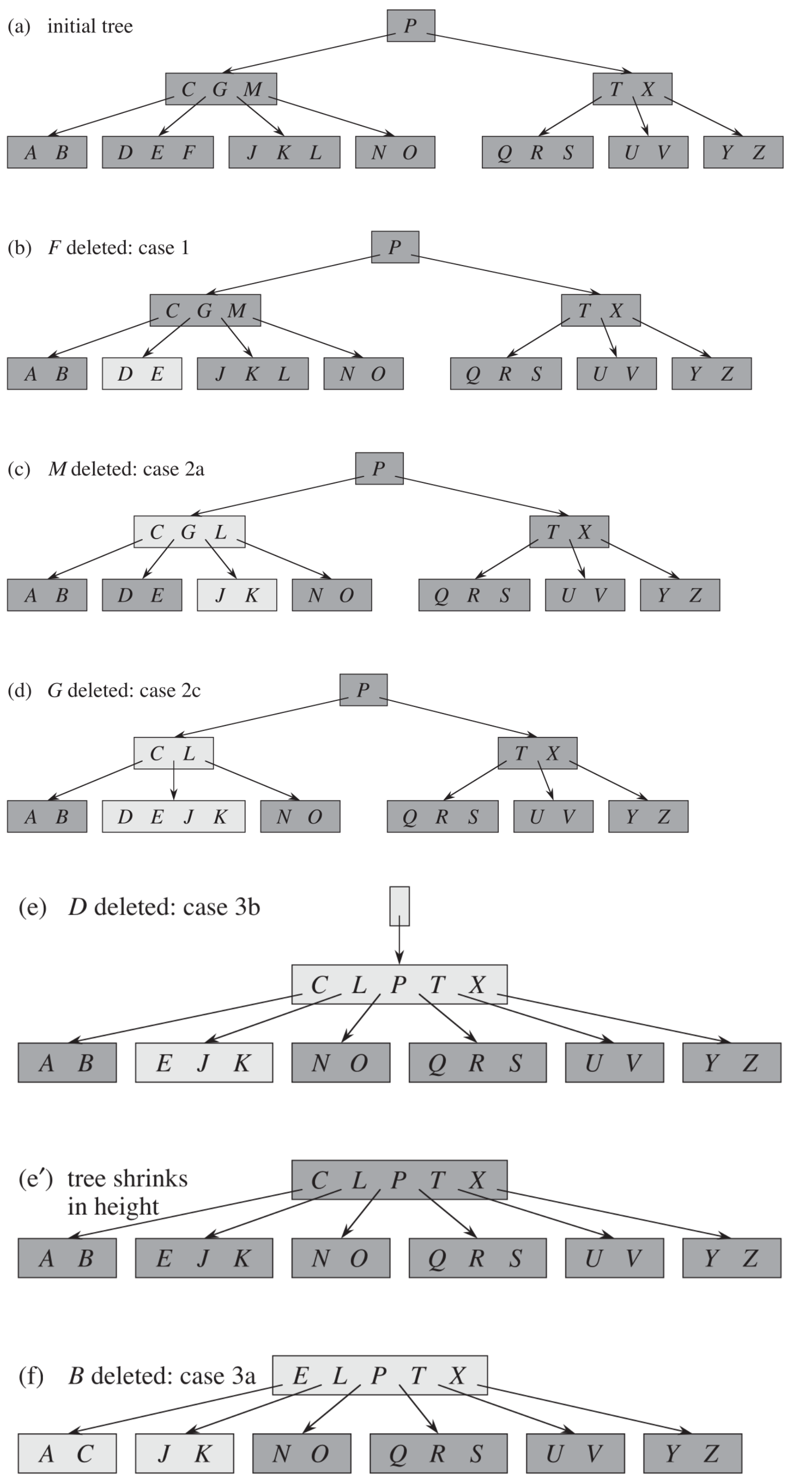
2.1.4 Deletion

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.