5. Greedy algoritm

5.0 Basics

5.1 Minimum spanning trees

5.2 Knapsack problem

5.3 Job sequencing with deadline

5.4 Optimal merge patterns

5.5 Huffman encoding

5.4 Optimal merge pattern

- Problem:
  - Merge k files → Various combinations of merging patterns
  - c.f. Merging two sorted files containing n and m records into one file takes O(n+m).
  - Determine an optimal way to pairwisely merge k sorted files together.

5.4 Optimal merge pattern

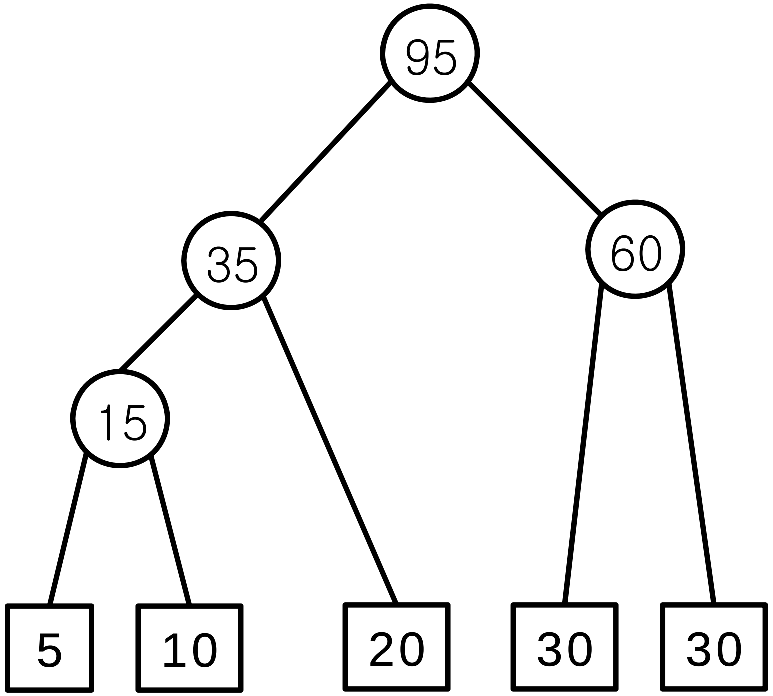
- Example:
  - X1, X2 and X3 are three sorted files of length 30, 20, and 10 records each.
  - Merge pattern 1:
    - Merge X1 & X2 → Y1 (50 steps)
    - Merge Y1 & X3 → Y2 (60 steps)
  - Merge pattern 2:
    - Merge X2 & X3 → Y1 (30 steps)
    - Merge Y1 & X1 → Y2 (60 steps)
  - Compare the time required

5.4 Optimal merge pattern

- Solution strategy:
  - 2-way merge pattern can be represented as a binary merge tree with minimum weighted external path length
  - Example:
    - Five patterns with (20, 30, 10, 5, 30)

5.4 Optimal merge pattern

- Solution strategy:
  - Corresponding binary tree



5.4 Optimal merge pattern

- Solution strategy:
  - If a pattern of length q_i is stored at a node whose depth is d_i, then the number of moves of the pattern is q_i d_i.
  - The total moves of the patterns is:

$$\sum_{1 \le i \le n} d_i q_i$$

  - The weighted external path length of a tree

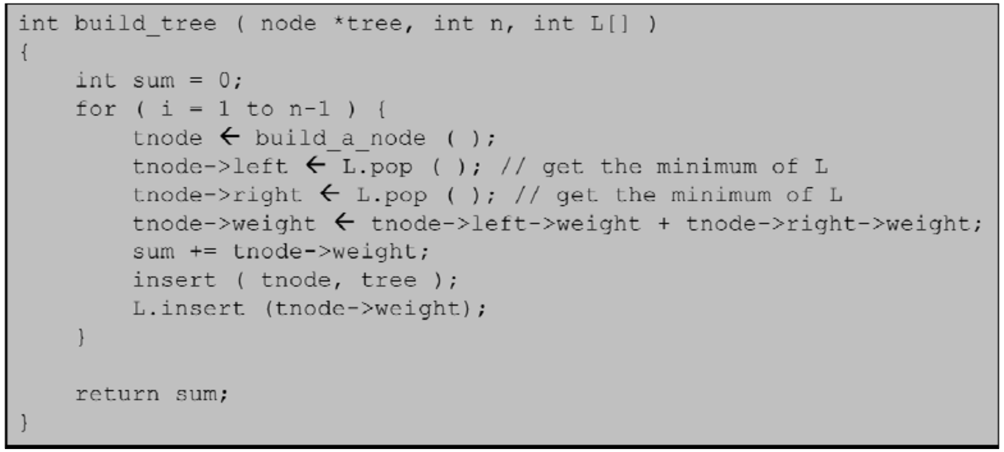
5.4 Optimal merge pattern

- Optimal merge pattern

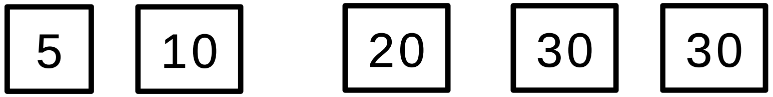
```
int build_tree ( node *tree, int n, int L[] )
{
    int sum = 0;
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← L.pop ( ); // get the minimum of L
        tnode->right ← L.pop ( ); // get the minimum of L
        tnode->weight ← tnode->left->weight + tnode->right->weight;
        sum += tnode->weight;
        insert ( tnode, tree );
        L.insert (tnode->weight);
    }

    return sum;
}
```

5.4 Optimal merge pattern

- Solution strategy:
  - Corresponding binary tree



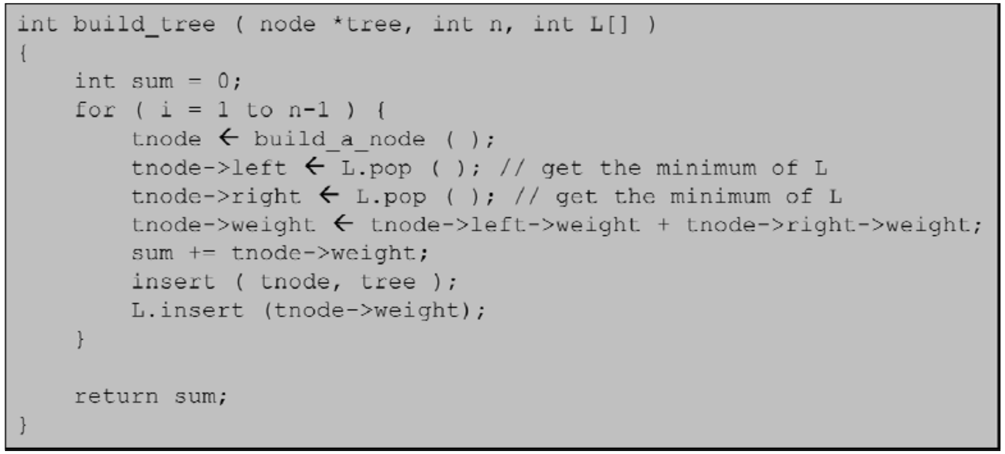
```
int build_tree ( node *tree, int n, int L[] )
{
    int sum = 0;
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← L.pop ( ); // get the minimum of L
        tnode->right ← L.pop ( ); // get the minimum of L
        tnode->weight ← tnode->left->weight + tnode->right->weight;
        sum += tnode->weight;
        insert ( tnode, tree );
        L.insert (tnode->weight);
    }

    return sum;
}
```



5 10 20 30 30

5.4 Optimal merge pattern

- Time complexity
  - L: priority queue
    - L.pop ( ) → O(1)
    - L.insert ( ) → O(log n)
  - O(n) * O(log n) → O(n log n)



```
int build_tree ( node *tree, int n, int L[] )
{
    int sum = 0;
    for ( i = 1 to n-1 ) {
        tnode ← build_a_node ( );
        tnode->left ← L.pop ( ); // get the minimum of L
        tnode->right ← L.pop ( ); // get the minimum of L
        tnode->weight ← tnode->left->weight + tnode->right->weight;
        sum += tnode->weight;
        insert ( tnode, tree );
        L.insert (tnode->weight);
    }

    return sum;
}
```