CSIEB0100 Data Structures

Lecture 09 Sorting II – External Sorting

Shiow-yang Wu 吳秀陽
Department of Computer Science and Information Engineering
National Dong Hwa University

Lecture material is mostly home-grown, partly taken from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

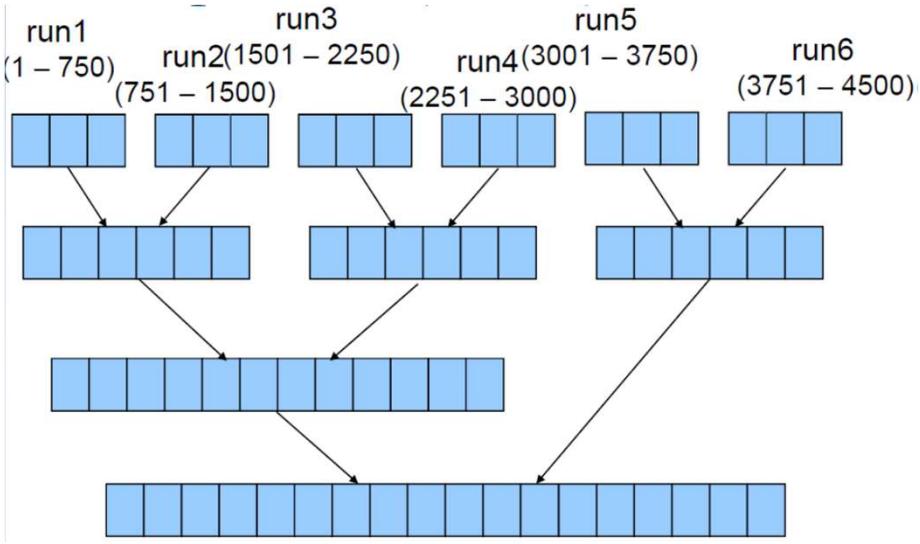
External Sorting

- Some lists are too large to fit in the memory of a computer. So internal sorting is not possible.
- Some records are stored in the disk. System retrieves a block of data from a disk at a time. A block contains multiple records.
- The most popular method for sorting on external storage devices is merge sort.
  - Segments of the input list are sorted.
  - Sorted segments (called runs) are written onto external storage.
  - Runs are merged until only one run is left.

External Sorting Example (7.12)

- Consider a computer which is capable of sorting 750 records is used to sort 4500 records.
- 6 runs are generated with each run sorting 750 records.
- Allocate three 250-record blocks of internal memory for performing merging runs. Two for input runs and the last one is for output.
- Three factors contributing to the read/write time of disk:
  - Seek time: move read/write head to the right cylinder.
  - Latency time: time until the right sector is under the head.
  - Transmission time: time to transmit data to/from disk

Example 7.12 (contd.)



Example 7.12 (contd.)

- $t_{IO} = t_s + t_l + t_{rw}$
  - t_s = maximum seek time
  - t_l = maximum latency time
  - t_{rw} = time to read/write on block of 250 records
- t_{IS} = time to internal sort 750 records
- $n \times t_m$ = time to merge n records from input buffers to the output buffer

Example 7.12 (contd.)

- Computing time of the external sort

	Operation	Time
(1)	Read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
(2)	Merge runs 1 to 6 in pairs	$36t_{IO} + 4500t_m$
(3)	Merge two runs for 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
(4)	Merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
	Total time	$132t_{IO} + 12000t_m + 6t_{IS}$

K-Way Merging

- To merge m runs via 2-way merging will need $\lceil \log_2 m \rceil + 1$ passes where m is the number of runs
- With k-way merge on m runs, we need $\lceil \log_k m \rceil$ passes over the data.
- If we use higher order merge, the number of passes as well as input/output time would be reduced.

K-Way Merging

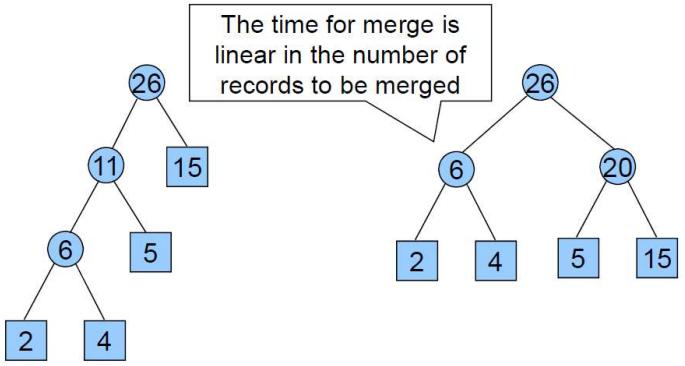
- But is it always true that the higher order of merging, the less computing time we will have?
  - Not necessary!
  - *k-1* comparisons are needed to determine the next output.
  - The number of key comparisons is $n(k-1)\log_k m$
  - If loser tree is used to reduce the number of comparisons, we can achieve $O(n \log_2 m)$ complexity
  - The data block size reduced as k increases. Reduced block size implies the increase of data passes (seek and latency times)

Buffer Handling for Parallel Operation

- To achieve better performance, multiple input buffers and two output buffers are used to avoid idle time.
- Evenly distributing input buffers among all runs may still have idle time problem. Buffers should be dynamically assigned to whoever needs to retrieve more data to avoid halting the computing process.
- We should take advantage of task overlapping and keep computing process busy and avoid idle time.

Optimal Merging of Runs

- The runs may not be of the same size. Since the merge time is linear to the run size, we optimize the merging of runs.



weighted external path length
= 2*3 + 4*3 + 5*2 + 15*1
= 43(=6+11+26)

weighted external path length
= 2*2 + 4*2 + 5*2 + 15*2
= 52=(6+20+26)

Optimal Merging of Runs (contd.)

- The cost of a *k*-way merge of *n* runs of length q_i, $1 \le i \le n$, is minimized by using a merge tree of degree *k* that has minimum weighted external path length.
- A good solution to the problem above has been given by D. Huffman. (next slide)

Huffman Code

- Assume we want to obtain an optimal set of codes for messages M_1, M_2, …, M_{n+1}. Each code is a binary string that will be used for transmission of the corresponding message.
- At receiving end, a decode tree is used to decode the binary string and get back the message.
- A zero is interpreted as a left branch and a one as a right branch. These codes are called Huffman codes.
- The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node.

Huffman Code (contd.)

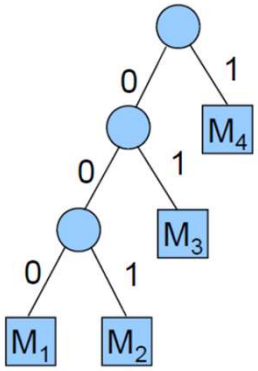
- If q_i is the relative frequency with which message M_i will be transmitted, then the expected decoding time is

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

where d_i is the distance of the external node for message M_i from the root node.

Huffman Code (contd.)

- The expected decoding time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length.



Code	Message
1	M_4
01	M_3
001	M_2
000	M_1

BinaryTree Class

```
class BinaryTreeNode {
friend BinaryTree;
private:
  int weight;
  BinaryTreeNode *LeftChild, *RightChild;
};

class BinaryTree {
public:
  int weight();
  BinaryTree(BinaryTree bt1, BinaryTree bt2) {
     root->LeftChild = bt1.root;
     root->RightChild = bt2.root;
     root->weight = bt1.root->weight + bt2.root->weight;
  };
private:
  BinaryTreeNode *root;
}
```


Huffman Function

```
void huffman(List<BinaryTree> l)
// l is a list of single node binary trees
{
   int n = l.Size(); // number of binary trees in l
   for (int i = 0; i < n-1 ; i++) { // loop n-1 times
       BinaryTree first = l.DeleteMinWeight();
       BinaryTree second = l.DeleteMinWeight();
       BinaryTree *bt = new BinaryTree(first, second);
       l.Insert(bt);
   }
}
```

Huffman Code Example

- Input text:

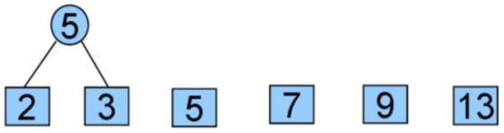
  ABBABCCDDE…
- Scan the text to calculate the number of appearances of each character

  A: 2, B: 3, C: 5, D: 7, E: 9, F: 13
- Build Huffman tree
- Encode text

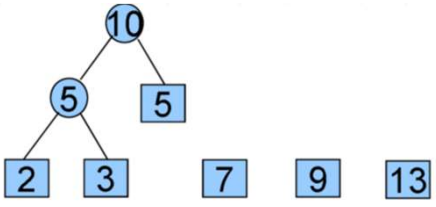
  1000 1001 1001 …

Huffman Code Example

{(A: 2), (B: 3), (C: 5), (D: 7), (E: 9), (F: 13)}



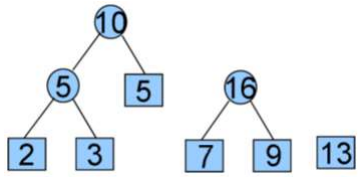
{(A, B: 5), (C: 5), (D: 7), (E: 9), (F: 13)}



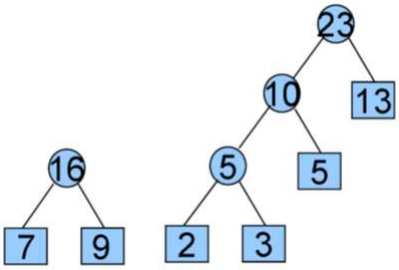
{(D: 7), (E: 9), (A,B,C: 10), (F: 13)}

Huffman Code Example

{(D: 7), (E: 9), (A,B,C: 10), (F: 13)}



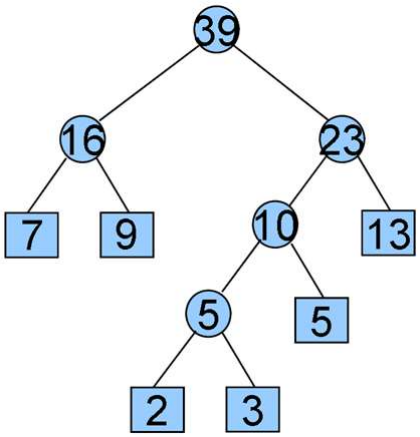
{(A,B,C: 10), (F: 13), (D, E: 16)}



{(D, E: 16), (A,B,C,F: 23)}

Huffman Code Example

{(D, E: 16), (A,B,C,F: 23)}

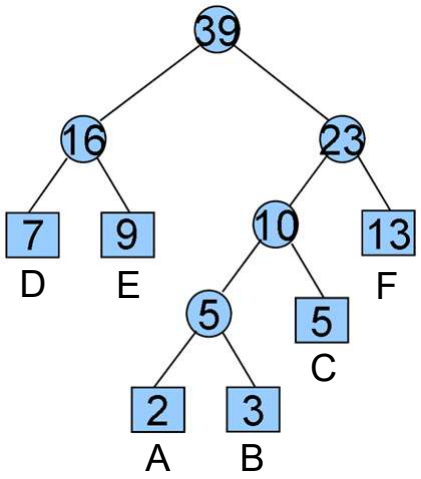


{(A, B, C, D, E, F: 39)}

Huffman Code Example

- The code table

Symbol	Code
A	1000
B	1001
C	101
D	00
E	01
F	11



Optimal Merging with Huffman Tree

- We can use Huffman Code Method to get the optimal merging order.
- Consider the run length q_i as the weight of run R_i.
- Combine all runs into a list.
- Call the Huffman function (p.16)
- The final tree is the optimal merge pattern.
- Try to devise an example and test it.
- It is more efficient to use a min-heap instead of a list. (how?)

Complexity Analysis

- In each iteration, two DeleMin and one Insert is performed. The total time is

  $T(n) = O(n-1) \times \max(O(\text{DelMin}), O(\text{Insert}))$

- If list is not sorted, then O(DelMin)=O(n), O(Insert)=1, so $T(n) = (n-1) \times n = O(n^2)$.
- If list is sorted in an array, then O(DelMin)=O(1), O(Insert)=O(n), so $T(n) = (n-1) \times n = O(n^2)$.
- If list is organized as a min-heap, then O(DelMin)=O(1), O(Insert)=O(logn), so $T(n) = (n-1) \times \log n = O(n \log n)$.