CSC3100 Data Structure

Name: Dai Yulong

Student ID: 119010054

Report of A4

[Question 1] Huffman Encoding

1. **What are the possible solutions for this problem?**

- The Data structure I used: (1) linked list (2) Priority Queue (3) array (4) Tree
- Other data structure that may help to increase program performance: (1) linked list (2) heap (3) Advanced tree
- I/O: (1) cin() and cout(). (2) scanf() and printf(). (3) getchar() and putchar() (4) buffer

2. How do I solve this problem?

- Initially, I used the sample code of priority queue in the 3002 class. It has basic constructor, destructor, enqueue, and dequeue functions. It used a linked list to store all elements. And the locations of elements in the linked list depend on their priorities. And priority depends on the frequency that it shows up. The class function will be as follows.

```
class PriorityQueue {
public:
   PriorityQueue();
   ~PriorityQueue();
   int size();
   void clear();
   void enqueue(Node* element);
   Node* dequeue();

private:
   struct Cell {
      Node *node;
      Cell *link;
   };

   Cell *head;   //Pointer to the cell at the head
   int count;    //Number of elements in the queue
};

PriorityQueue::PriorityQueue() {
   head = new Cell;
   head->link = NULL;
   head->node = NULL;
   count = 0;
}

PriorityQueue::~PriorityQueue() {
   clear();
}

int PriorityQueue::size() {
   return count;
}

void PriorityQueue::clear() {
   while (count > 0) {
      dequeue();
   }
}
```

```
void PriorityQueue::enqueue(Node *n)
{
    Cell *cp = new Cell;
    Cell *temp = head;
    cp->node = n;
    if (count == 0) {
        head->link = cp;
        cp->link = NULL;
    } else {
        while (true) {
            if ((temp->link != NULL) && ((n->freqency > temp->link->node->freqency)
                temp = temp->link;
                continue;
            } else {
                break;
            }
        }
        cp->link = temp->link;
        temp->link = cp;
    }
    count++;
}

Node *PriorityQueue::dequeue()
{
    Node *n = head->link->node;
    head->link = head->link->link;
    count--;
    return n;
}
```

- Then, I declare some data structures:

(1) Node (“code” to store Huffman code. “key” to store a, b, c, d…. “right” is the right child, “left” is the left child. “frequency” is the times that char appears in the string) (2) An array “arr” is to store all different nodes.

```
struct Node {
    string code = "";
    Node* left = NULL;
    Node* right = NULL;
    char key;
    int freqency;
};
```

```
PriorityQueue pq;
Node *arr[256];
for (int i = 0; i < 256; i++) {
    arr[i] = NULL;
}

int pos1 = 0;
for (int i = 0; i < (int)s.length(); i++) {
    int pos2 = 0;
    for (int j = 0; j < pos1; j++) {
        if (arr[j]->key == s[i]) {
            break;
        } else {
            pos2 += 1;
            continue;
        }
    }
    if (pos2 == pos1) {
        Node *n=new Node;
        n->key = s[i];
        n->freqency = 1;
        arr[pos1] = n;
        pos1++;
    } else {
        arr[pos2]->freqency += 1;
    }
}
```

- Then I build a huff:

(1) First, I enqueue all the nodes in the “arr” to the priority queue (2) Then, I pop the first two nodes and add their frequency. (3) Create a new node and set two nodes as its right child and left child, respectively, and set the frequency (3) add “0” to first node’s each “code” of children string Huffman code and “1” to second node’s each “code” (including themselves).

```
for (int i = 0; i < pos1; i++) {
    Node *temp = arr[i];
    pq.enqueue(temp);
}

while (pq.size() >= 2) {
    Node *temp1 = pq.dequeue();
    Node *temp2 = pq.dequeue();
    Node *cp = new Node;
    cp->left = temp1;
    cp->right = temp2;
    cp->key = temp1->key;
    cp->freqency = temp1->freqency + temp2->freqency;

    add(temp1, "0");
    add(temp2, "1");
    pq.enqueue(cp);
}

string res ="";
for (int i = 0; i < (int)s.length(); i++) {
    for (int j = 0; j < 256; j++) {
        if (arr[j]->key == s[i]) {
            res += arr[j]->code;
            break;
        }
    }
}
```

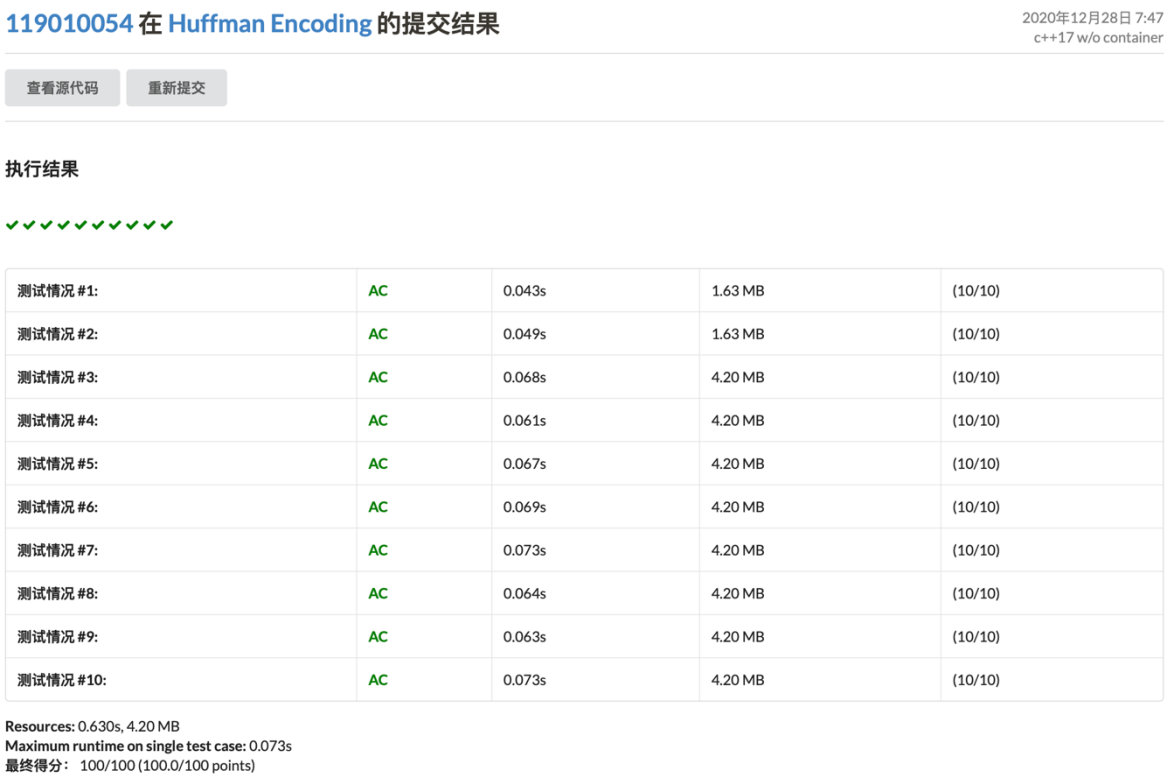
- Using “cout” to output the result.

3. How do I optimize your solution?

- I use an array instead of inked list to store all nodes so that I can efficiently find the element.
- I use priority queue to find the high-frequency elements. And it will be quicker.
- Actually, I use a node to store all properties of elements. This makes my program more readable and has higher performance.

4. Why is my solution better than others?

- I use priority queue. Thus, I can get the node with smallest frequency in O(1), but use. array it will cost O(n). Also, for finding the smallest node, Priority queue is more concise.
- I use linked list instead of an array in the priority queue. Since when push a. node into a Priority queue, linked list is much better than array.
- I use an array to store all nodes since I can get it with O(1). And I use a node to store all properties of an element, which will be more readable. Every time I only need to pop two elements and push one element. It will save lots of time.
- Results:



119010054 在 Huffman Encoding 的提交结果

2020年12月28日 7:47
c++17 w/o container

查看源代码 重新提交

执行结果

✔✔✔✔✔✔✔✔✔✔

测试情况 #1:	AC	0.043s	1.63 MB	(10/10)
测试情况 #2:	AC	0.049s	1.63 MB	(10/10)
测试情况 #3:	AC	0.068s	4.20 MB	(10/10)
测试情况 #4:	AC	0.061s	4.20 MB	(10/10)
测试情况 #5:	AC	0.067s	4.20 MB	(10/10)
测试情况 #6:	AC	0.069s	4.20 MB	(10/10)
测试情况 #7:	AC	0.073s	4.20 MB	(10/10)
测试情况 #8:	AC	0.064s	4.20 MB	(10/10)
测试情况 #9:	AC	0.063s	4.20 MB	(10/10)
测试情况 #10:	AC	0.073s	4.20 MB	(10/10)

Resources: 0.630s, 4.20 MB
Maximum runtime on single test case: 0.073s
最终得分： 100/100 (100.0/100 points)

5. How do I test my program?

- To check the correctness of the program initially, I produce some data. Data includea some special case: (1) char with the same frequency (2) just 1 char input (3) space and other chars
- To check the efficiency, I put the program to OJ to test default data.

6. What are the possible further improvements?

- Use heap to implement the Priority queue instead of using a linked list. This will make the time complexity of “enqueue” from O(n) to O(logn).
- Use getchar and putchar instead of cin and cout.
- The way I use to store Huffman code is not efficient (string). We can use an array to store the digit. Or use bool to represent “0”/”1” (since bool takes less memory than string)

[Question 2] Dijkstra

1. **What are the possible solutions for this problem?**

- The Data structure I used: (1) linked list (2) priority queue (3) array (4) Tree (5) heap (6) map.
- Other data structure that may help to increase program performance: (1) matrix (2) advanced tree
- I/O: (1) cin() and cout(). (2) scanf() and printf(). (3) getchar() and putchar() (4) buffer

2. How do I solve this problem?

- I use getchar() and putchar() for input and output.
- Then I create two structures in C++, which called “Node” and “neighbor” respectively. “Node” stores the ID and the distance from the start node. “Neighbor” stores the id, the path weight with the previous node, and a link pointed to the previous node.
- Thirdly, I define an array called “array” to store all the pointers that pointed to neighbors. The index of this array is the node’s Id. This structure looks like a hash map. Every pointer in the “array” points to a linked list consisting of neighbors.

```
144.     Neighbor* array[n];
145.     int results[n];
146.     bool arr1[n];
147.     for (int i = 0; i < n; i++) {
148.         arr1[i] = 1;
149.     }
150.     for (int i=0;i<n;i++) {
151.         array[i] = NULL;
152.     }
153.
154.     for (int j=0;j<m;j++) {
155.         int num1, num2;
156.         int32_t weight;
157.         num1 = input();
158.         num2 = input();
159.         weight = input();
160.         Neighbor* cp = new Neighbor;
161.         cp->id = num2-1;
162.         cp->weight = weight;
163.         cp->link = array[num1-1];
164.         array[num1-1] = cp;
165.     }
166.
```

- I also declare a class called “priority queue”. It used heap structure to get sort the elements. And it has constructor, destructor, extract_min, min_heapify and push function.
- Also, I create an array called arr1 of type bool. It stores all elements that have been visited. And the later procedure will skip this point.

- Then for the algorithm – Dijkstra. It's in the main function. Once a node pops from extract_min, it will be pop out and relax the neighbors that haven't been visited. The relax function detects the neighbors of that node and push the node into a priority queue.

3. How do I optimize your solution?

- At first, I didn't use the priority queue. I use an array to store all nodes and another array to store the location of each node in the first array. And it will cost lots of extra time and spaces. Then I used priority queue through the linked list. It will also cost too much time. Initially, I used four arrays as follows:

```
struct Node arr1[1000001];    //Used heap sort
struct Neighbor **arr2 = new struct Neighbor* [1000001];   //Store all the neighbors of a node
int32_t arr3[1000001];   //The key of every node
int arr4[1000001];
```

Finally, I changed to priority queue through heap.

- First, I use cin and cout to input and output numbers. And it will cost too much time. Then I use scanf() and printf(). The output is still not ideal. At last, I use putchar() and getchar().

4. Why is my solution better than others?

- The method: getchar() and putchar() I used are much efficient
- I use a structure that is similar to hash map to store all neighbors. It's convenient to get the nodes' distance with their neighbors. And it takes much less memory than matrix.
- Priority Queue can use O(1) time to get the smallest node.
- Results:

执行结果

✔×14

测试情况 #1:	AC	0.073s	1.63 MB	(0/0)
测试情况 #2:	AC	0.048s	1.63 MB	(0/0)
测试情况 #3:	AC	0.046s	3.08 MB	(0/0)
测试情况 #4:	AC	0.069s	6.53 MB	(0/0)
测试情况 #5:	AC	0.330s	39.69 MB	(10/10)
测试情况 #6:	AC	0.544s	70.89 MB	(10/10)
测试情况 #7:	AC	0.998s	76.82 MB	(10/10)
测试情况 #8:	AC	0.261s	53.68 MB	(10/10)
测试情况 #9:	AC	0.218s	56.43 MB	(10/10)
测试情况 #10:	AC	0.367s	91.51 MB	(10/10)
测试情况 #11:	AC	0.669s	114.46 MB	(10/10)
测试情况 #12:	AC	0.673s	28.09 MB	(10/10)
测试情况 #13:	AC	0.688s	28.09 MB	(10/10)
测试情况 #14:	AC	0.717s	70.63 MB	(10/10)

Resources: 5.700s, 114.46 MB
Maximum runtime on single test case: 0.998s
最终得分： 100/100 (100.0/100 points)

5. How do I test my program?

- To check the correctness of the program initially, I randomly generate 1,000,000 data. To test the performance of my program
- To check efficiency, I also use OJ to test default data.

6. What are the possible further improvements?

- We can consider use matrix to store the edges. It will save lots of time
- Using an array to store the location of each element. Then we can distinctly swap them, which will be more efficient.
- Reduce the numbers of structure in C++. I used two structures, which are “node” and “neighbor”, the improvement can be using just one structure as a container that store all the information.
- Using some advanced data structure or algorithms, i.e.forward-star.