DS 3^{rd} homework

58121102 Jiang Yuchu

30 September 2022

P_{138} 1. Extend the stack ADT by adding functions to output a stack; split a stack into two stacks with one containing the bottom half elements and the second the remaining elements; and to combine two stacks into one by placing all elements of the second stack on top of those of the first stack. Write C++ code for your new functions.

Answer:

The following code implements the function of dividing a stack into two.

```
template <class T>
Stack<T> Stack<T>::Split() {
    Stack<T> res, rev;
    for (int size = (top + 1) / 2; size--;) {
        rev.Push(Top());
        Pop();
    }
    while (!rev.Empty()) {
        res.Push(Top());
        rev.Pop();
    }
    return res;
}
```

The following code implements the function of merge two stacks into one.

```
template <class T>
void Stack<T>::Merge(Stack<T> src) {
    Stack<T> rev;
    for (int size = src.top + 1; size--;) {
        rev.Push(src.Top());
        src.Pop();
    }
    while (!rev.Empty()) {
        Push(rev.Top());
        rev.Pop();
    }
}
```

P_{138} 2. Consider the railroad switching network given in Figure 33 Railroad cars numbered 1, 2, 3,..., n are initially in the top right track segment (in this order, left to right): Railroad cars can be moved into the vertical track segment one at a time from either of the horizontal segments and then moved from the vertical segment to any one of the horizontal segments. The vertical segment operates as a stack as new cars enter at the top and cars depart the vertical segment from the top. For instance, if $n = 3$, we could move car I into the vertical segment, move 2 in, move 3 in, and then take the cars out producing the new order 3, 2, 1. For $n = 3$ and 4 what are the possible permutations of the cars that can be obtained? Are any permutations not possible?

Answer:

① For $n = 3$, all permutations other than 3, 1, 2 are possible (7 in total).
② For $n = 4$, 3 4 1 2, 3 1 2 4, 3 1 4 2 and all permutations starting with 4 except 4321 are impossible.

P_{147} 1. Rewrite functions *Push* and *Pop* (Programs 3.10 and 3.12) using the variable *lastOp* as discussed in this section. The queue should now be able to hold up to capacity elements. The complexity of each of your functions should be $\Theta(1)$(exclusive of the time taken to double queue capacity when needed).

Answer:

```
enum class opType { Push, Pop }
template <class T>
void Queue<T>::Push(const T& value) {
    if ((rear % capacity) == front) {
        if (lastOp == OpType::Push)
            doubling();
    }
    rear = (rear + 1) % capacity;
    lastOp = opType::Push;
    queue[rear] = value;
}
template <class T>
void Queue<T>::Pop() {
    if (isEmpty()) throw "Queue␣is␣empty.␣Cannot␣delete.";
    if ((rear % capacity) == front) {
        if (lastOp == OpType::Push)
            doubling();
    }
    front = (front + 1) % capacity;
    lastOp = opType::Pop;
    queue[front].~T();
}
```

P_{147} 3.To the class *Queue*,add a function to split a queue into two queues.The first queue is to contain every other element beginning with the first;the second queue contains the remaining elements.The relative order of queue elements is unchanged.What is the complexity of your function?

Answer:

```
template<class T>
void Queue<T>::divide(Queue<T>& res1, Queue<T>& res2) const {
    int _front = front, _rear = rear;
    for (int i = 0; (temp + 1) % capacity != _rear; ++i) {
        if ((i + 1) & 1)
            res1.Push(queue[(_front + 1) % capacity]);
        else
            res2.Push(queue[(_front + 1) % capacity]);
        _front = (_front + 1) % capacity;
    }
}
```

It's obvious that the time complexity of this function is $O(n)$, where n is the queue length.

P_{157} 2. What is the maximum path length from start to finish for any maze of dimensions $m \times p$?

Answer:
I guess the complexity should be $O(m \times p)$? I'm not sure.

P_{157} 3. Write and test recursive version of Path.What is the time complexity of your recursive version?

Answer:

See the code in p135-3.cpp. Similar to the non-recursive version, its time complexity should also be $m \times p$.

Experiment

P_{168}2.

1 Experiment target

1.1 Background

Simulate an airport landing and takeoff pattern. The airport has three runways, runway 1, runway2, and runway 3. There are four landing holding patterns, two for each of the first two runways. Arriving planes will enter one of the holding pattern queues, where the queues are to be as close in size as possible. When a plane enters a holding queue, it is assigned an integer id number and an integer giving the number of time units the plane can remain in the queue before it must land (because of low fuel level). There is also a queue for takeoffs for each of the three runways. Planes arriving in a takeoff queue are also assigned an integer id. The takeoff queues should be kept approximately the same size.

At each time, up to three planes may arrive at the landing queues and up to three planes may arrive at the takeoff queues. Each runway can handle one takeoff or landing at each time slot. Runway 3 is to be used for takeoffs except when a plane is low on fuel. At each time unit, planes in either landing queue whose air time has reached zero must be given priority over other landings and takeoffs. If only one plane is in this category, runway 3 is to be used. If more than one, then the other runways are also used (at each time, at most three planes can be serviced in this way).

Use successive even (odd)integers for id's of planes arriving at takeoff (landing)queues. At each time unit assume that arriving planes are entered into queues before takeoffs or landings occur. Try to design your algorithm so that neither landing nor takeoff queues grow excessively. However, arriving planes must be placed at the ends of queues. Queues cannot be reordered.

1.2 Goals

The output should clearly indicate what occurs at each time unit. Periodically output

(a) the contents of each queue;

(b) the average takeoff waiting time;

(c) the average landing waiting time;

(d) the average flying time remaining on landing;

(e) the number of planes landing with no fuel reserve.

(b)and (c)are for planes that have taken off or landed, respectively. The output should be self-explanatory and easy to understand (and uncluttered).

2 Basic idea

2.1 Data structure

First of all, these are some global variables.

```
1 size_t id_in = 0, id_out = 1; // generate id of each plane
2 size_t current_time           = 0;
3 size_t emergency_landing_cnt  = 0; // goal (e)
4 size_t takeoff_n_per_unittime = 0; // goal (b)
```

```
5  size_t takeoff_wait_time_sum  = 0;
6  size_t landing_n_per_unittime = 0; // goal (c) and (d)
7  size_t landing_wait_time_sum  = 0;
8  size_t landing_left_time_sum  = 0;
```

There are 2 kinds of planes. Planes to landing are inherit from planes to takeoff, because planes to landing need more fields to record info.

```
1  struct Plane {
2      size_t id;
3      size_t wait_time;
4      // other functions
5  };
6
7  struct LandingPlane : public Plane {
8      size_t left_time;
9      // other functions
10 };
11 using TakeoffPlane = Plane;
```

2 kinds of runways are designed. The first two runways mentioned in the background are inherit from the third runway(called emergency runway). Notice that I used a priority queue to represent landing queue, which can help me to arrange those planes which have little left time to come to the front. Besides, `handler` is used to process planes that have little left time in waiting queue.

The `MaintainGuard` is a complex class which used RAII. It is aims to maintain info, such as `left_time`, `wait_time`, and handle the situation where a plane needs to land immediately.

```
1  class RunwayBase {
2  public:
3      class MaintainGuard;
4
5      //other functions
6  private:
7      queue<TakeoffPlane> takeoff; // takeoff queue
8      priority_queue<LandingPlane,
9          vector<LandingPlane>,
10         greater<>> landing; //landing queue
11     unique_ptr<Plane> current = nullptr; //current plane
12
13     function<void(const LandingPlane&)> handler;
14
15
16 };
17 class Runway : public RunwayBase {
18     queue<LandingPlane> wait1, wait2; //wait landing queue
19     // other functions
20 };
21 using EmergencyRunway = RunwayBase;
```

Finally, here is class `Airport`.

```
1  class Airport {
2      // other functions
3  private:
4      array<unique_ptr<RunwayBase>, 3> arr; // runways
5
6      priority_queue<LandingPlane> emergency_planes;
7      queue<TakeoffPlane> takeoff_planes;
8
```

```
 9      ofstream queue_info; // write content of queues per unit time
10      ofstream stuff_info; // write info mentioned in goals
11  };
```

2.2 Algorithm procedure

I'd like to list some code before I illustrating.

```
 1  class RunwayBase {
 2  public:
 3      struct MaintainGuard;
 4
 5      virtual void run(size_t* landing_cnt = nullptr);
 6
 7      // dispatch planes to takeoff/landing queue
 8      // in both functions, each plane will get an id
 9      virtual bool dispatch(const LandingPlane& plane);
10      void dispatch(const TakeoffPlane& plane);
11
12      // other functions
13  };
14
15  class Runway : public RunwayBase {
16  public:
17      // add landing planes to waiting queue
18      // it ensures that the size of waiting queues is kept close to
19      void addLandingPlane(const LandingPlane& plane);
20
21      bool dispatch(const LandingPlane& plane) override;
22
23      void run(size_t* landing_cnt) override;
24
25      // other functions
26  };
27
28  class Airport {
29  public:
30      void addPlane(const TakeoffPlane& plane);
31      void addPlane(const LandingPlane& plane);
32
33      array<unique_ptr<Plane>, 3> run();
34
35      //other functions
36  };
37
38  int main() {
39      //others...
40      Airport airport;
41      uniform_int_distribution<> u(2, 20);
42      random_device              rd;
43      for (auto i : views::iota(0, 25)) {
44          airport.addPlane(TakeoffPlane{ 0 });
45          size_t left_time = u(rd);
46          // others...
47          airport.addPlane(LandingPlane(0, left_time));
48      }
49
50      while (!airport.empty()) {
```

```
51            auto [rw1, rw2, em] = move(airport.run());
52            // output result
53        }
54    }
```

My algorithm can be divided into the following steps:

1. Add planes to airport.
2. Invoke function `airport.run()` to dispatch planes repeatly. This function can be divided into:
   (a) Dispatch emergency planes(if exist) to each runway. Emergency runway has the highest priority, followed by other two. But in total, the size of takeoff queue of three ways keeps close.
   (b) Dispatch takeoff plane to runway that has the smallest takeoff queue. Due to limitations of background(At each time, up to three planes may arrive at the landing queues and up to three planes may arrive at the takeoff queues), dispatching will be repeated for only 3 times.
   (c) Invoke function `run` of each runway. In this step, the proper plane will be dispatched to the runway. If `landing.top().left_time == 1`, then do landing for sure. Otherwise, dispatch plane between `landing.top()` and `takeoff.front()` according to their id(which indicates the order of their arrival to takeoff/landing queue).
   (d) Record relevant info and return the results.
3. Output results.

3 Experimental procedure

1. Generate 25 takeoff planes and 25 landing planes, then add them to airport.
2. Invoke function `airport.run()` in a loop until there are no planes at the airport.
3. Record results.

4 Results and analysis

This is a screenshot of the running.

```
time 1:
runway1:0       runway2:2       emergency:5
time 2:
runway1:1       runway2:3       emergency:4
time 3:
runway1:6       runway2:11      emergency:10
time 4:
runway1:9       runway2:15      emergency:14
time 5:
runway1:8       runway2:16      emergency:18
time 6:
runway1:13      runway2:19      emergency:20
time 7:
runway1:21      runway2:23      emergency:26
time 8:
runway1:24      runway2:27      emergency:30
time 9:
runway1:25      runway2:22      emergency:12
time 10:
runway1:28      runway2:34      emergency:32
time 11:
runway1:31      runway2:33      emergency:36
time 12:
runway1:37      runway2:38      emergency:42
time 13:
runway1:41      runway2:39      emergency:7
time 14:
runway1:45      runway2:40      emergency:17
time 15:
runway1:empty   runway2:43      emergency:44
time 16:
runway1:empty   runway2:47      emergency:29
time 17:
runway1:empty   runway2:49      emergency:35
```

Results of Goal (a) is placed in *queue_info.csv*, and results of goal (b) (e) are placed in *stuff_info.csv*. The left times of landing planes that generated these data are be placed in *left_time.csv*.