Simulations

1. Circular Queues
   - buffer of fixed capacity
   - using the STL deque

2. Simulating Waiting Lines using Queues
   - modeling arrival times
   - generating and processing jobs
Simulations

1 Circular Queues
   - buffer of fixed capacity
   - using the STL deque

2 Simulating Waiting Lines using Queues
   - modeling arrival times
   - generating and processing jobs
Two main types of sequences:

1. array or vector: subscripting operator \[ \] ;
2. linked list: nodes connected by pointers.

Adapting access, push and pop:

- stack: last in first out
- queue: first in first out
a circular buffer

```
<table>
<thead>
<tr>
<th>front</th>
<th>back</th>
</tr>
</thead>
<tbody>
<tr>
<td>'e'</td>
<td>'a'</td>
</tr>
<tr>
<td>'a'</td>
<td>'b'</td>
</tr>
<tr>
<td>'b'</td>
<td>'c'</td>
</tr>
<tr>
<td>'c'</td>
<td>'d'</td>
</tr>
<tr>
<td>'d'</td>
<td>'e'</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>front</th>
<th>back</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>'b'</td>
</tr>
<tr>
<td>'b'</td>
<td>'c'</td>
</tr>
<tr>
<td>'c'</td>
<td>'d'</td>
</tr>
<tr>
<td>'d'</td>
<td>'e'</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
arrays and linked lists

For the implementation, we can use

1. an array (or vector): with fixed or variable capacity;
2. a linked list: either single or doubly linked.

Advantages and disadvantages of arrays:

+ consecutive items in memory give fast access;
- either fixed capacity or $O(n)$ to reallocate.

Advantages and disadvantages of lists:

+ memory efficient, all operations are $O(1)$;
- pointers occupy space, no consecutive storage.

A combined approach: linked list of arrays.
arrays of arrays

To amortize the reallocation cost, use an array of arrays:

Two stage allocation scheme:

- declare an array of pointers,
- allocate memory chunks as needed.
matrix of characters

$ /tmp/array_of_arrays
  abcdefg
  hjiklmn
  opqrstuv
#include <iostream>
using namespace std;
#define CAPACITY 5
#define BLOCK_SIZE 7

int main()
{
    char *A[CAPACITY];

    A[0] = new char[BLOCK_SIZE];
    for(int i=0; i<BLOCK_SIZE; i++)
        A[0][i] = 'a' + i;
    A[1] = new char[BLOCK_SIZE];
    for(int i=0; i<BLOCK_SIZE; i++)
        A[1][i] = 'a' + BLOCK_SIZE + i;
    for(int i=0; i<BLOCK_SIZE; i++)
        A[2][i] = 'a' + 2*BLOCK_SIZE + i;
for(int i=0; i<3; i++)
{
    for(int j=0; j<BLOCK_SIZE; j++)
        cout << " " << A[i][j];
    cout << endl;
}

for(int i=0; i<3*BLOCK_SIZE; i++)
{
    cout << "A[" << i << "] = ";
    int j = i / BLOCK_SIZE;
    int k = i % BLOCK_SIZE;
    cout << "A[" << j << "][" << k << "] = ";
    cout << A[j][k] << endl;
}
Simulations

1. Circular Queues
   - buffer of fixed capacity
   - using the STL deque

2. Simulating Waiting Lines using Queues
   - modeling arrival times
   - generating and processing jobs
the deque

Maintaining an index to front and back, we arrive at a double ended queue, or *deque*.

Operations `pop_front()` and `pop_back()` remove item at front and at end of queue respectively.
Simulating waiting line:

- `pop_front()` : customer at front is served and leaves,
- `pop_back()` : customer at end quits queue.

Operations `push_back(t)` and `push_front(t)` append item $t$ at end or at front of queue respectively.
We can use deque either as queue or as stack.
using the STL deque

#include <iostream>
#include <deque>
using namespace std;

int main()
{
    deque<char> q;
    for(int i=0; i<5; i++)
        q.push_back('a'+i);
    cout << "our deque :";
    for(int i=0; i<5; i++)
        cout << " " << q[i];
    cout << endl;
}
using an iterator

deque<char> q;

for(int i=0; i<5; i++)
    q.push_back('a'+i);

cout << "using an interator :";
deque<char>::iterator j = q.begin();
while(j!= q.end())
    cout << " " << *j++;
cout << endl;
Simulations

1. Circular Queues
   - buffer of fixed capacity
   - using the STL deque

2. Simulating Waiting Lines using Queues
   - modeling arrival times
   - generating and processing jobs
the Poisson distribution

Let $\lambda > 0$, be a positive real number.

$\lambda$ = expected number of events in a time interval assuming events occur independently

The probability that exactly $k$ events happen:

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}.$$  

$\lambda$ is average, deviation is $\sqrt{\lambda}$
modeling arrivals

Assume: on average 4 arrivals per minute. We model 20 minutes:

$ /tmp/random_poisson_numbers
give parameter lambda : 4
give number of samples : 20
  3 3 2 1 5 4 5 4 3 6 2 4 3 3 2 7 7 4 5 3
average : 3.8
frequencies : 0 1 3 6 4 3 1 2 0

Interpretation of 20 minutes of simulation:
- sometimes we see as few as 1 person,
- sometimes there are as many as 7 arrivals.
generating Poisson numbers

Input: $\lambda$, average #events per time interval.
Output: $N$, #events per time interval.

$L := e^{-\lambda}; k := 0; p := 1;$
do
  $k := k + 1;$
  draw $u \in [0, 1]$ at random;
  $p := p \times u;$
while ($p > L$);
return $N := k - 1$.

Donald E. Knuth: *The Art of Computer Programming*, volume 2
#include <cstdlib>
#include <ctime>
#include <cmath>
#include <iostream>
using namespace std;

double random_double()
{
  int r = rand();
  double f = (double) r;
  return (f/RAND_MAX);
}
random Poisson number generator

double poisson_rand( double lambda )
{
    double L = exp(-lambda);
    int k = 0;
    double p = 1.0;

    do
    {
        k = k + 1;
        double u = random_double();
        p = p * u;
    }
    while (p > L);

    return (k-1);
}
#ifndef POISSON_NUMBER_GENERATOR_H
#define POISSON_NUMBERGENERATOR_H

class Poisson_Number_Generator
{
    public:

    Poisson_Number_Generator( double lambda );
    // initializes seed of generator

    int sample();
    // returns a random number

    private:

    double lambda_parameter;

};
#endif
#include "poisson_number_generator.h"
#include <cstdlib>
#include <ctime>
#include <cmath>

Poisson_Number_Generator::
    Poisson_Number_Generator( double lambda )
{
    lambda_parameter = lambda;
    srand(time(0));
}
the `sample()` method

```cpp
int Poisson_Number_Generator::sample()
{
    double L = exp(-lambda_parameter);
    int k = 0;
    double p = 1.0;

    do
    {
        k = k + 1;
        int r = rand();
        double u = double(r)/RAND_MAX;
        p = p * u;
    } while (p > L);

    return (k-1);
}
```
```cpp
#include "poisson_number_generator.h"
#include <iostream>
using namespace std;

int main()
{
    cout << "give parameter lambda : ";
    double lambda; cin >> lambda;

    Poisson_Number_Generator p(lambda);

    cout << "give number of samples : ";
    int n; cin >> n;
```
```cpp
int d = 2*int(lambda);
int freq[d+1];
for (int i=0; i<=d; i++) freq[i] = 0;

int sum = 0;
for(int i=0; i<n; i++)
{
    int r = p.sample();
    cout << " " << r;
    sum = sum + r;
    freq[(r > d ? d : r)]++;
}
cout << endl;
cout << "average : " << double(sum)/n << endl;
cout << "frequencies :";
for(int i=0; i<=d; i++) cout << " " << freq[i];
cout << endl;
```
Simulations

1. Circular Queues
   - buffer of fixed capacity
   - using the STL deque

2. Simulating Waiting Lines using Queues
   - modeling arrival times
   - generating and processing jobs
generating jobs

$ /tmp/simulate_waiting
give number of minutes : 10
give number of arrivals per minute : 3
give maximum size of each job : 5
at t = 0 : 4 arrivals with #items 4 1 5 5
at t = 1 : 2 arrivals with #items 2 4
at t = 2 : 7 arrivals with #items 2 5 4 3 5 4 2
at t = 3 : 2 arrivals with #items 5 5
at t = 4 : 1 arrivals with #items 1
at t = 5 : 3 arrivals with #items 4 4 2
at t = 6 : 4 arrivals with #items 4 5 4 5
at t = 7 : 3 arrivals with #items 2 3 4
at t = 8 : 4 arrivals with #items 1 5 1 4
at t = 9 : 2 arrivals with #items 4 5
processing jobs

at $t = 0$ : 4 arrivals with #items 4 1 5 5
at $t = 1$ : 2 arrivals with #items 2 4

If processor can handle $\geq 15$ items/minute, then jobs arriving at $t = 1$ have no wait.

If processor can handle 5 items/minute, then first job at $t = 1$ has to wait 2 minutes.

Given #items/minute processor can handle, what is the average waiting time?
data & subroutines

We use a deque of jobs: `deque<Job>`, where

```c
struct Job
{
    int arrival; // arrival time in minutes
    int size;    // size of job
};
```

Three functions:

1. `generate_jobs`: returns a queue, given parameters;
2. `write_jobs`: information for each job + summary;
3. `process_jobs`: compute total waiting time.

Introduction to Data Structures (MCS 360)
Queue Applications
L-18 9 October 2017 28 / 33
generating jobs

deque<Job> generate_jobs( int n, int m, double L )
{
    Poisson_Number_Generator p(L);
    deque<Job> q;

    for(int i=0; i<n; i++)
    {
        int r = p.sample();
        for(int j=0; j<r; j++)
        {
            Job b;
            b.arrival = i;
            b.size = 1 + rand() % m;
            q.push_back(b);
        }
    }
    return q;
}
writing jobs

```cpp
void write_jobs( deque<Job> q )
{
    deque<Job>::iterator i = q.begin();
    int ind = 0;
    int sum = 0;
    while( i != q.end() )
    {
        Job j = *i++;
        cout << "job " << ind++ << " arrived at " << j.arrival \\
             << " has size " << j.size << endl;
        sum = sum + j.size;
    }
    cout << "queue has " << ind \\
         << " jobs and total #items " << sum \\
         << endl;
}
```
int process_jobs( deque<Job> q, int h )
{
    int wait = 0;    // total waiting time
    int elapsed = 0; // elapsed minutes
    int ready = h;   // #items able to process

    for(int i=0; !q.empty(); i++,q.pop_front())
    {
        Job j = q.front();
        if(ready < h)
            if(j.arrival > elapsed) ready = h;
        if(j.arrival >= elapsed)
            cout << " no wait" << endl;
        else
        {
            int w = elapsed - j.arrival;
            cout << " wait " << w << " minutes\n";
            wait = wait + w;
        }
    }
}
computing elapsed time

```c
int work = j.size;
if (work > ready)
{
    work = work - ready;
    elapsed++; ready = h;
}
while (work > ready)
{
    work = work - ready; elapsed++;
}
ready = ready - work;
if (ready == 0)
{
    elapsed++; ready = h;
}
return wait;
```
Summary + Exercises

Ended Chapter 6 with array of arrays and simulations.

Exercises:

1. Consider the code to simulate a waiting line to make a queue of jobs with size of each jobs between 1 and 8. Give code to split the queue in two: one with small jobs (sizes 1 or 2) and another with larger jobs (size > 2).

2. Write a function `pop_at_random()` on a deque to return a random item from the deque and to remove it.

3. Describe the changes to the simulation for multiple processors.