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

MCS 360 Lecture 18
Introduction to Data Structures
Jan Verschelde, 24 February 2020
1. **Circular Queues**
   - buffer of fixed capacity
   - using the STL deque

2. **Simulating Waiting Lines using Queues**
   - modeling arrival times
   - generating and processing jobs
sequential data structures

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

Introduction to Data Structures (MCS 360)
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:

```
  S  S  S  S  S
  |   |   |   |   |
  |   |   |   |   |
  |   |   |   |   |
  |   |   |   |   |
```

Two stage allocation scheme:
- declare an array of pointers,
- allocate memory chunks as needed.
$ \text{array_of_arrays}
\begin{align*}
\text{a} & \quad \text{b} & \quad \text{c} & \quad \text{d} & \quad \text{e} & \quad \text{f} & \quad \text{g} \\
\text{h} & \quad \text{i} & \quad \text{j} & \quad \text{k} & \quad \text{l} & \quad \text{m} & \quad \text{n} \\
\text{o} & \quad \text{p} & \quad \text{q} & \quad \text{r} & \quad \text{s} & \quad \text{t} & \quad \text{u}
\end{align*}
\begin{align*}
\text{A}[0] &= \text{A}[0][0] = \text{a}, & \text{A}[1] &= \text{A}[0][1] = \text{b} \\
\text{A}[2] &= \text{A}[0][2] = \text{c}, & \text{A}[3] &= \text{A}[0][3] = \text{d} \\
\text{A}[4] &= \text{A}[0][4] = \text{e}, & \text{A}[5] &= \text{A}[0][5] = \text{f} \\
\text{A}[6] &= \text{A}[0][6] = \text{g}, & \text{A}[7] &= \text{A}[1][0] = \text{h} \\
\text{A}[8] &= \text{A}[1][1] = \text{i}, & \text{A}[9] &= \text{A}[1][2] = \text{j} \\
\text{A}[10] &= \text{A}[1][3] = \text{k}, & \text{A}[11] &= \text{A}[1][4] = \text{l} \\
\text{A}[12] &= \text{A}[1][5] = \text{m}, & \text{A}[13] &= \text{A}[1][6] = \text{n} \\
\text{A}[14] &= \text{A}[2][0] = \text{o}, & \text{A}[15] &= \text{A}[2][1] = \text{p} \\
\text{A}[16] &= \text{A}[2][2] = \text{q}, & \text{A}[17] &= \text{A}[2][3] = \text{r} \\
\text{A}[18] &= \text{A}[2][4] = \text{s}, & \text{A}[19] &= \text{A}[2][5] = \text{t} \\
\text{A}[20] &= \text{A}[2][6] = \text{u}
\end{align*}
#include <iostream>
define CAPACITY 5
define BLOCK_SIZE 7
using namespace std;

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
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 waiting line.

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

```cpp
#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 iterator :";
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
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 $\sqrt{\lambda}$. 

the Poisson distribution
modeling arrivals

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

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

Donald E. Knuth: *The Art of Computer Programming*, volume 2
random doubles

```cpp
#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);
}
Object-Oriented Encapsulation

```cpp
#ifndef __POISSON_NUMBERGENERATOR_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 <cstdlib>
#include <ctime>
#include <cmath>
#include "poisson_number_generator.h"

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);
}
```
#include <iostream>
#include "poisson_number_generator.h"
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;
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

$ 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?
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.
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;
}
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;
}
flowchart for the simulation

\[
W = [0]; b = P[0]; i = 1
\]

\[
i < \text{len}(A)\?
\]

\[
\text{if False then}
\text{return } W
\]

\[
t = A[i] - A[i-1]
\]

\[
\text{if } t \geq b \text{ then}
\text{if True then}
\text{b = 0}
\text{else}
\text{b = b - t}
\text{else}
\text{W.append(b)}
\]

\[
b = b + P[i]
\]

\[
i = i + 1
\]
processing jobs

```cpp
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

```cpp
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 remove it.

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