# *Data Structures*
# *Lec-4 Queues and Hashing*

# Review: Program Complexities

- Algorithms
- Asymptotic Notation
- Asymptotic Performance
- Analyze program complexities

# Review: Algorithms

- What is an algorithm?

  A sequence of elementary computational steps that transform the input into the output

- What for?

  A tool for solving well-specified computational problems, e.g., Sorting, Matrix Multiplication

- What do we need to do with an algorithm?
  - Correctness Proof:

    for every input instance, it halts with the correct output
  - Performance Analysis (1 second or 10 years?):

    How does the algorithm behave as the problem size gets large

    both in running time and storage requirement

# Review: Correctness of Algorithm

- Why can the algorithm correctly sort?
- We only consider algorithms with loops
  - ➢ Find a property as loop invariant
- How to show something is loop invariant?
  - ➢ Initialization:

    It is true prior to the first iteration of the loop
  - ➢ Maintenance:

    If it is true before an iteration, it remains true before the next iteration
  - ➢ Termination:

    When the loop terminates, the invariant gives a useful property that helps to show the algorithm is correct

# Review: A Sorting Problem

Input : $\langle a_0, a_1, \ldots, a_{n-1} \rangle$

Output: A permutation (re-ordering) $\langle a'_0, a'_1, \ldots, a'_{n-1} \rangle$ of the input sequence such that $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$

For example:

*<22, 51, 34, 44, 67, 11>* becomes *<11, 22, 34, 44, 51, 67>*

# Review: Insertion Sort

***5***, 3, 1, 2, 6, 4

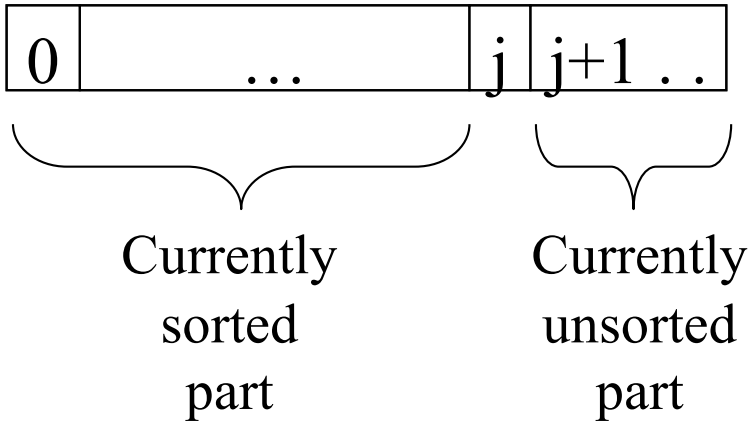
***3, 5***, 1, 2, 6, 4

***1, 3, 5***, 2, 6, 4

***1, 2, 3, 5***, 6, 4

***1, 2, 3, 5, 6***, 4

***1, 2, 3, 4, 5, 6***




- To sort *A[0,1,…,n-1]* in place
- Steps:
  - Pick element A[j]
  - Move A[j-1,…,0] to the right until proper position for A[j] is found
- Example 1 3 5 2 6 4

# Review: Insertion Sort (cont.)

**Insertion-Sort (A)**

```
1. for j=1 to n-1
2.       key=A[j]
3.       i=j-1
4.       while i>=0 and A[i]>key
5.                 A[i+1]=A[i]
6.                 i=i-1
7.       A[i+1]=key
```

| | A[0] | A[1] | A[2] | A[3] | A[4] | A[5] |
|---|---|---|---|---|---|---|
| j=1 | 5 | 3 | 1 | 2 | 6 | 4 |
| j=2 | 3 | 5 | 1 | 2 | 6 | 4 |
| j=3 | 1 | 3 | 5 | 2 | 6 | 4 |
| j=4 | 1 | 2 | 3 | 5 | 6 | 4 |
| j=5 | 1 | 2 | 3 | 5 | 6 | 4 |
| j=6 | 1 | 2 | 3 | 4 | 5 | 6 |

j=3

1 3 5 2 6 4

1 3 5→2 6 4

1 3→2 5 6 4

1 2 3 5 6 4

# Review: Running time of Insertion Sort

```
Insertion-Sort(A)
1  for j = 1 to n-1
2      key = A[j]
3      i = j-1
4      while i >= 0 and A[i] > key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

| Cost | times |
|---|---|
| $c_1$ | **n** |
| $c_2$ | **n-1** |
| $c_3$ | **n-1** |
| $c_4$ | $\Sigma_{j=1..n-1} (t_j+1)$ |
| $c_5$ | $\Sigma_{j=1..n-1} t_j$ |
| $c_6$ | $\Sigma_{j=1..n-1} t_j$ |
| $c_7$ | **n-1** |

**$c_1$, $c_2$, .. = running time for executing line 1, line 2, etc.**

**$t_j$ = no. of times that line 5,6 are executed, for each j.**

The running time T(n)

$= c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1} (t_j+1))+c_5*(\Sigma_{j=1..n-1} t_j)+c_6*(\Sigma_{j=1..n-1} t_j)+c_7*(n-1)$

# Review: Analyzing Insertion Sort

$$T(n) = c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1}\ (t_j+1))+ c_5*(\Sigma_{j=1..n-1}\ t_j)+c_6*(\Sigma_{j=1..n-1}\ t_j)+c_7*(n-1)$$

**Worse case:**

Reverse sorted: for example, 6,5,4,3,2,1

➔ inner loop body executed for all previous elements.

➔ $t_j=j$.

➔ $$T(n) = c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1}\ (j+1))+ c_5*(\Sigma_{j=1..n-1}\ j)+c_6*(\Sigma_{j=1..n-1}\ j)+c_7*(n-1)$$

➔ $$T(n) = An^2+Bn+C$$

Note: $\Sigma_{j=1..n-1}\ j = n(n-1)/2$

$\Sigma_{j=1..n-1}\ (j+1) = (n+2)(n-1)/2$

# Review: Analyzing Insertion Sort

$T(n) = c_1*n+c_2*(n-1)+c_3*(n-1)+c_4*(\Sigma_{j=1..n-1}\ (t_j+1))$
$+c_5*(\Sigma_{j=1..n-1}\ t_j)+c_6*(\Sigma_{j=1..n-1}\ t_j)+c_7*(n-1)$

**Worst case** Reverse sorted → inner loop body executed for all previous elements. So, $t_j=j$.

→ T(n) is quadratic: $T(n)=An^2+Bn+C$

**Average case** Half elements in A[0..j-1] are less than A[j]. So, $t_j = j/2$

→ T(n) is also quadratic: $T(n)=A'n^2+B'n+C'$

**Best case** Already sorted → inner loop body never executed. So, $t_j=0$.

→ T(n) is linear: $T(n)=An+B$

# Review: Kinds of Analysis

**(Usually) Worst case Analysis:**

T(n) = max time on any input of size ***n***

Knowing it gives us a guarantee about the upper bound.

In some cases, worst case occurs fairly often

**(Sometimes) Average case Analysis:**

T(n) = average time over all inputs of size ***n***

Average case is often as bad as worst case.

**(Rarely) Best case Analysis:**

Cheat with slow algorithm that works fast on some input.

Good only for showing bad lower bound.

# Review: Order of Growth

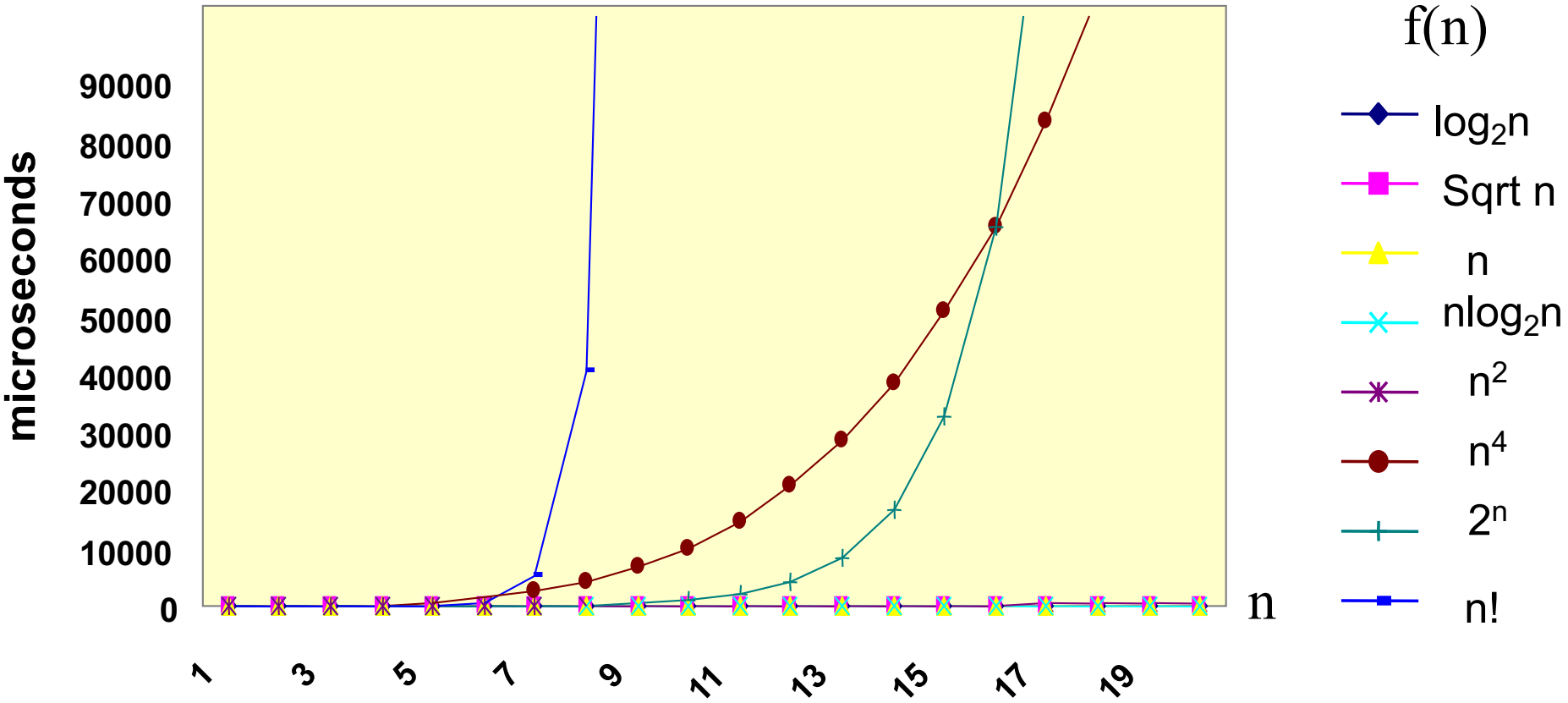
Examples:

Running time of algorithm in microseconds (in term of data size *n*)

| | f(n) | n=20 | n=40 | n=60 |
|---|---|---|---|---|
| **Algorithm A** | $Log_2n$ | $4.32 * 10^{-6}$sec | $5.32 * 10^{-6}$sec | $5.91 * 10^{-6}$sec |
| **Algorithm B** | Sqrt(n) | $4.47 * 10^{-6}$sec | $6.32 * 10^{-6}$sec | $7.75 * 10^{-6}$sec |
| **Algorithm C** | n | $20 * 10^{-6}$sec | $40 * 10^{-6}$sec | $60 * 10^{-6}$sec |
| **Algorithm D** | $n \log_2n$ | $86 * 10^{-6}$sec | $213 * 10^{-6}$sec | $354 * 10^{-6}$sec |
| **Algorithm E** | $n^2$ | $400 * 10^{-6}$sec | $1600 * 10^{-6}$sec | $3600 * 10^{-6}$sec |
| **Algorithm F** | $n^4$ | 0.16 sec | 2.56 sec | _____ sec |
| **Algorithm G** | $2^n$ | 1.05 sec | 12.73 days | _____ years |
| **Algorithm H** | n! | 77147 years | $2.56 * 10^{34}$ years | $2.64 * 10^{68}$ years |

# Review: Order of Growth

Assume: an algorithm can solve a problem of size *n* in f(*n*) microseconds ($10^{-6}$ seconds).




Note: for example, for all f(n) in $\Theta(n^4)$,

the shapes of their curves are nearly the same as $f(n)=n^4$.

# Review: Asymptotic Notation

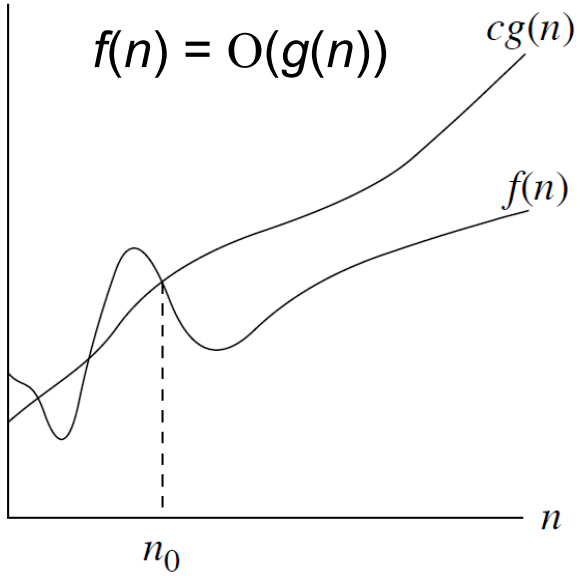
- How can we indicate running times of algorithms?
- Need a notation to express the growth rate of a function
- A way to compare "size" of functions:
  - $O$-notation ("Big-oh") $\approx \leq$ (upper bound)
  - $\Omega$-notation ("Big-omega") $\approx \geq$ (lower bound)
  - $\Theta$-notation ( "theta") $\approx =$ (sandwich)

# Review: $O$ -notation (1/2)

- $O$-notation provides an **asymptotic upper bound** of a function.
- For a given function $g(n)$, we denote $O(g(n))$

(pronounced "big-oh" of $g$ of $n$) by the set of functions:

$O(g(n))$ = { $f(n)$: there exist **positive** constants $c$ and $n_0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }



# Review: $O$ -notation (2/2)

- We write $f(n) = O(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $O(g(n))$
  - Give that $g(n)$ is an upper bound for $f(n)$ to within a constant factor

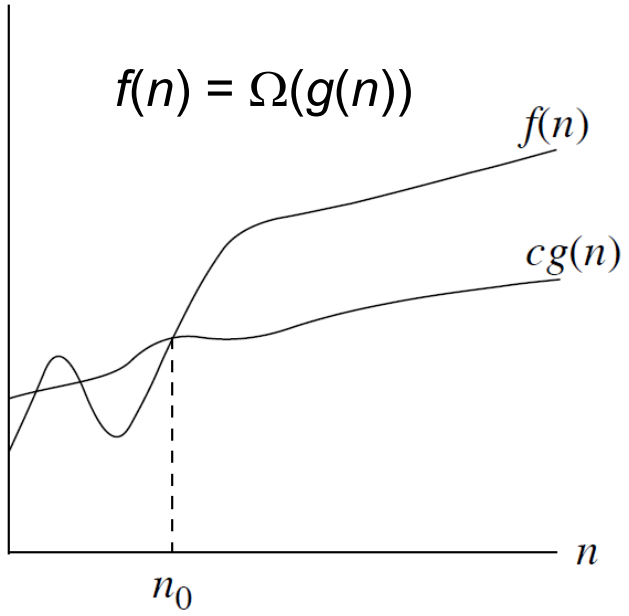
- Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$
  - When $n = 1$: $2(1)^2 = 2 \leq (1)^3 = 1$ ☒
  - When $n = 2$: $2(2)^2 = 8 \leq (2)^3 = 8$ ☑
  - When $n = 3$: $2(3)^2 = 18 \leq (3)^3 = 27$ ☑

# Review: $\Omega$ -notation (1/2)

- $\Omega$-notation provides an **asymptotic lower bound** of a function.
- For a given function $g(n)$, we denote $\Omega(g(n))$

(pronounced "big-omega" of $g$ of $n$) by the set of functions:

$\Omega(g(n))$ = { $f(n)$: there exist positive constants $c$ and $n_0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }



# Review: $\Omega$ -notation (2/2)

- We write $f(n) = \Omega(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $\Omega(g(n))$
  - Give that $g(n)$ is a lower bound for $f(n)$ to within a constant factor

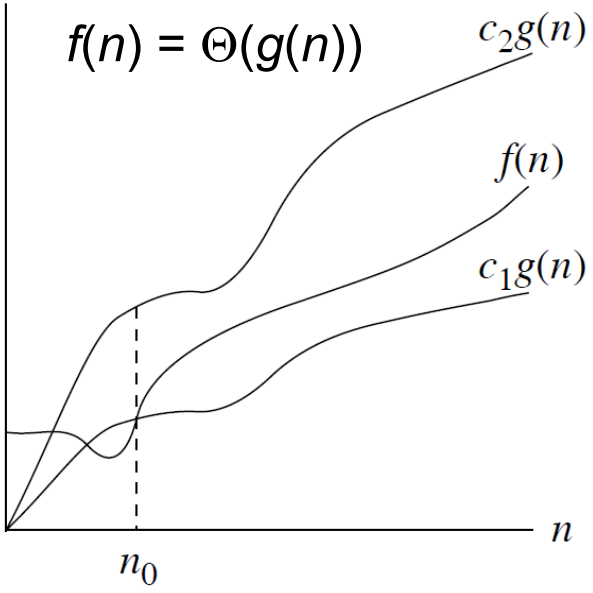
- Example: $n^2 + n = \Omega(n^2)$, with $c = 1$ and $n_0=1$
  - When $n = 1$: $(1)^2 + 1 = 2 \geq (1)^2 = 1$ ☑
  - When $n = 2$: $(2)^2 + 2 = 6 \geq (2)^2 = 4$ ☑
  - When $n = 3$: $(3)^2 + 3 = 12 \geq (3)^2 = 9$ ☑

# Review: $\Theta$ -notation (1/2)

- $\Theta$-notation provides an **asymptotically tight bound** of a function.
- For a given function $g(n)$, we denote $\Theta(g(n))$

(pronounced "theta" of $g$ of $n$) by the set of functions:

$\Theta(g(n)) = \{f(n)$: there exist positive constants $c_1$, $c_2$ and $n_0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$



# Review: $\Theta$ -notation (2/2)

- Theorem

  **$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$**

- Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$
  - When $n = 7$: $1/4[(7)^2] = 12.25 \leq (7)^2/2 - 2(7) = 10.5 \leq 1/2[(7)^2] = 24.5$ ☒
  - When $n = 8$: $1/4[(8)^2] = 16 \leq (8)^2/2 - 2(8) = 16 \leq 1/2[(8)^2] = 32$ ☑
  - When $n = 9$: $1/4[(9)^2] = 20.25 \leq (9)^2/2 - 2(9) = 22.5 \leq 1/2[(9)^2] = 40.5$ ☑

# Review: O versus o

- Little-o Notation
  - f(n) = o(g(n)): a strict upper bound for a function f(n)
  - o(g(n)) = { f(n): there exist positive constants c and n0 such that $0 \leq f(n) < cg(n)$ for all $n \geq n0$ }
  - o versus O : o means better e.g. $n=o(n^2)$
- Why 100n=O(n)?
  - When n is very big like 10000000000000000
  - 100n: 1000000000000000000
  - n: 10000000000000000
  - nearly the same
- Why $n=o(n^2)$?
  - When n is 10000000000000000
  - $n^2$ is 100000000000000000000000000000000
  - Differ a lot!

# Review: Asymptotic Notation

- Relationship between typical functions
  - $\log n = o(n)$
  - $n = o(n \log n)$
  - $n^c = o(2^n)$ where $n^c$ may be $n^2$, $n^4$, etc.
  - If $f(n)=n+\log n$, we call *log n* lower order terms

$\log n < \text{sqrt}(n) < n < n\log n < n^2 < n^4 < 2^n < n!$

# Review: Asymptotic Notation

- When calculating asymptotic running time
  - Drop low-order terms
  - Ignore leading constants
- Example 1: $T(n) = An^2+Bn+C$
  - $An^2$
  - $T(n) = O(n^2)$
- Example 2: $T(n) = Anlogn+Bn^2+Cn+D$
  - $Bn^2$
  - $T(n) = O(n^2)$

# Review: Asymptotic Performance

Very often the algorithm complexity can be observed directly from simple algorithms

```
Insertion-Sort(A)
1    for j = 1 to n-1
2         key = A[j]
3         i = j-1
4         while i >= 0 and A[i] > key
5              A[i+1] = A[i]
6              i = i - 1
7         A[i+1] = key
```

$O(n^2)$

There are 4 very useful rules for such Big-Oh analysis ...

# Review: Asymptotic Performance

**General rules for Big-Oh Analysis:**

### Rule 1. FOR LOOPS

The running time of a *for* loop is at most the running time of the statements inside the *for* loop (including tests) times no. of iterations

```
for (i=0;i<N;i++)
      a++;
```

O(N)

### Rule 2. NESTED FOR LOOPS

The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            k++;
```

$O(N^2)$

### Rule 3. CONSECUTIVE STATEMENTS

Count the maximum one.

```
for (i=0;i<N;i++)
      a++;

for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            k++;
```

$O(N^2)$

### Rule 4. IF / ELSE

For the fragment:

```
If (condition)
      S1
else
      S2,
```

take the test + the maximum for S1 and S2.

# Review: Asymptotic Performance

- Recursion

```
int Power(int base,int pow)
{       if (pow==0)   return 1;
        else   return base*Power(base,pow-1);
}
```

- Example

  $3^2=9$

  Power(3,2)=3*Power(3,1)

  Power(3,1)=3*Power(3,0)

  Power(3,0)=1

  T(n): the number of multiplications needed to compute Power(3,n)

  T(n)=T(n-1)+1; T(0)=0

  T(n)=n

  Function T(n) is O(n)

# Review: Tower of Hanoi

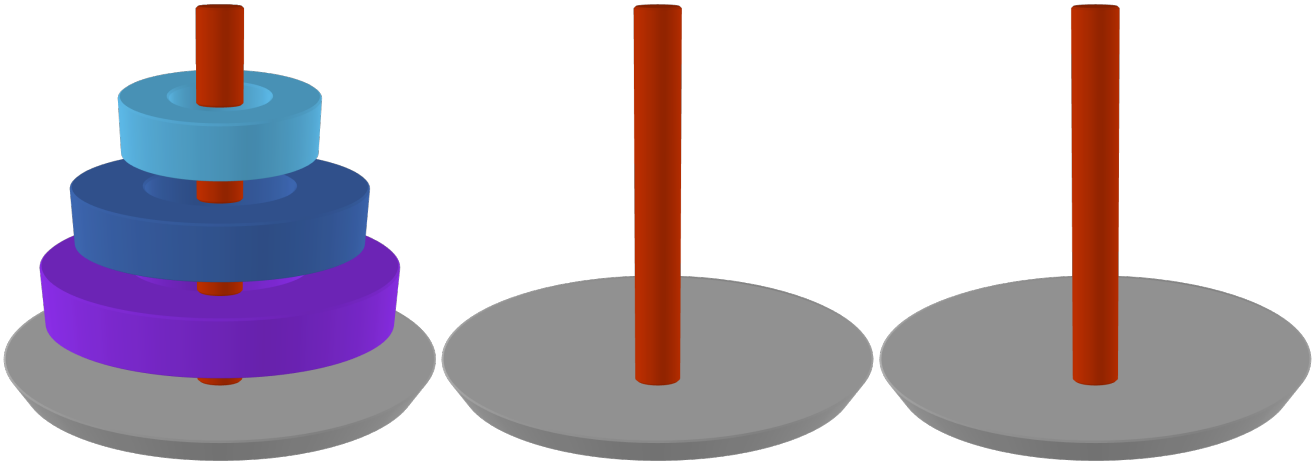
The problem:
Use fewest steps to move all disks from the source rod to the target without violating the rules through the whole process
(given one intermediate rod for buffering)?

Given some rods for stacking disks.

Rules:

(1) The disks must be stacked in order of size.

(2) Each time move 1 disk.



a source rod  an intermediate rod  a target rod

# Review: Tower of Hanoi

```
void Towers (int n, int Source, int Target, int Interm)
{
  if (n==1)
        cout<<“From”<<Source<<“To”<<Target<<endl;
  else
  {     Towers(n-1, Source, Interm, Target);
        Towers(1, Source, Target, Interm);
        Towers(n-1, Interm, Target, Source);
  }
}
```

How many “cout” are executed?

- T(n)=2T(n-1)+1

# Review: Recursive Relation

- $T(n)=T(n-1)+A; T(1)=1$
  - → $T(n)=O(n)$
- $T(n)=T(n-1)+n; T(1)=1$
  - → $T(n)=O(n^2)$
- $T(n)=2T(n/2) + n; T(1)=1$
  - → $T(n)=O(n \log n)$, why???
- More general form: $T(n)=aT(n/b)+cn$
  - Master's Theorem (You are not required to know)

$T(n) = 2T(n/2) + n$
$= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n + n$
$= 4(2T(n/8) + n/4) + 2n + n = 8T(n/8) + 4n + 2n + n$
$= \ldots$
$= ???\ T(1) + ??? + \ldots + 4n + 2n + n$

# Review Exercise

Give an analysis of the running time (Big-Oh will do).

```
(1)  sum = 0;
     for( i = 0; i < n; ++i )
         ++sum;
(2)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < n; ++j )
             ++sum;
(3)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < n * n; ++j )
             ++sum;
(4)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < i; ++j )
             ++sum;
(5)  sum = 0;
     for( i = 0; i < n; ++i )
         for( j = 0; j < i * i; ++j )
             for( k = 0; k < j; ++k )
                 ++sum;
(6)  sum = 0;
     for( i = 1; i < n; ++i )
         for( j = 1; j < i * i; ++j )
             if( j % i == 0 )
                 for( k = 0; k < j; ++k )
                     ++sum;
```

# Exercise 1

1. Suppose T1(N) = O(f(N)) and T2(N) = O(f(N)).

Which of the following are true?

a. T1(N) +T2(N)=O(f(N))

b. T1(N) − T2(N) = o(f(N))

c. T1(N) / T2(N) = O(1)

d. T1(N) = O(T2(N))

2. Order the following functions according to Big-O notation, i.e., if f(n) is O(g(n)) then f(n) should come before g(n).

6nlogn;

2^100;

loglogn;

log^2 n;

2^logn

# Exercise 2

1. What is the time complexity of the following code:

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

(a)

```
int a = 0, i = N;
while (i > 1) {
    a += i;
    i /= 2;
}
```

(b)

```
for(var i=1;i<n;i++)
    i*=k
```

(c)

# Objective

## Queue

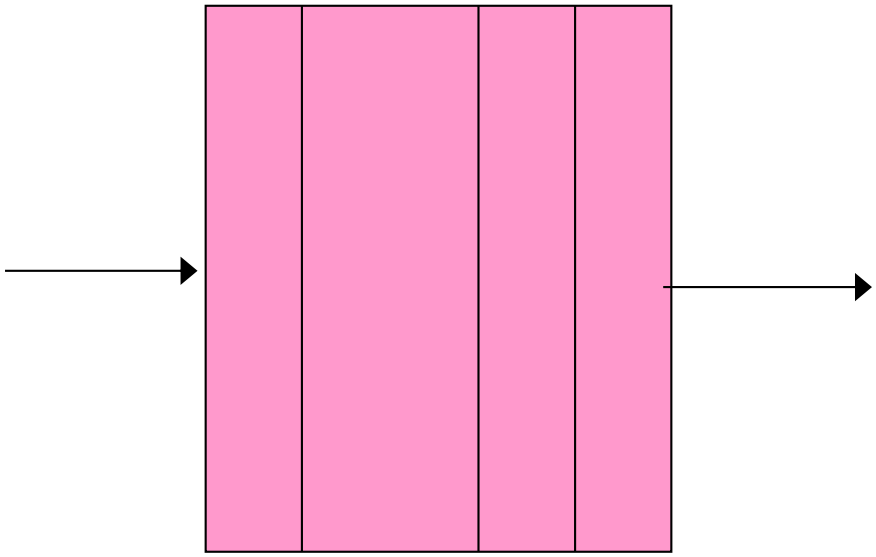
- Queue Abstract Data Type
- Sequential Allocation
- Linked Allocation
- Applications
- Priority Queues

## Hashing

- Sparse Data
- Key Based Data
- Hash Table
- Hash Functions
- Collision Resolution
- Applications

# Queue

- Queue is a list with the restriction that insertions are performed at one end and deletions are performed at the other end of the list
- Also known as: First-in-first-out (FIFO) list



# ADT of Queue

## Value:

A sequence of items that belong to some data type ITEM_TYPE

## Operations on q:

1. Boolean IsEmpty()
   Postcondition: If the queue is empty, return true, otherwise return false
2. Boolean IsFull()
   Postcondition: If the queue is full, return true, otherwise return false
3. ITEM_TYPE Dequeue() /*take out the front one and return its value*/
   Precondition: q is not empty
   Postcondition: The front item in q is removed from the sequence and returned
4. Void Enqueue(ITEM_TYPE e) /*to append one item to the rear of the queue*/
   Precondition: q is not full
   Postcondition: e is added to the sequence as the rear one

# ADT of Queue

## Value:

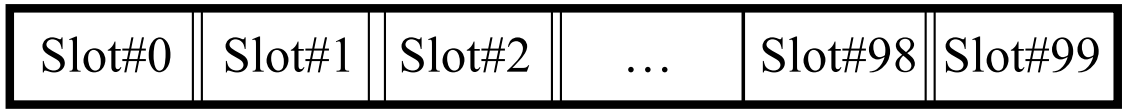
A sequence of items that belong to some data type ITEM_TYPE

## Operations on q:

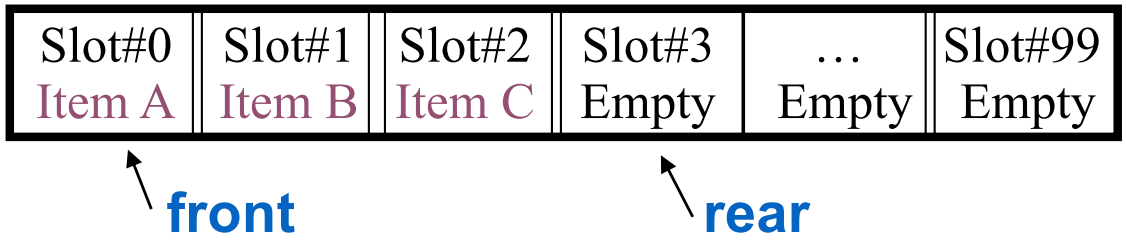
1. Boolean IsEmpty()
   Postcondition: If the queue is empty, return true, otherwise return false
2. Boolean IsFull()
   Postcondition: If the queue is full, return true, otherwise return false
3. ITEM_TYPE Dequeue() /*take out the front one and return its value*/
   Precondition: q is not empty
   Postcondition: The front item in q is removed from the sequence and returned
4. Void Enqueue(ITEM_TYPE e) /*to append one item to the rear of the queue*/
   Precondition: q is not full
   Postcondition: e is added to the sequence as the rear one

# Implementation of Queue Sequential Allocation (Using Array)

```
#define TOTAL_SLOTS 100
class MyQueue
{
        private:
                int front; //the index of the front slot that contains the front item
                int rear;  //the index of the first empty slot at the rear of queue
                int items[TOTAL_SLOTS];
};
```

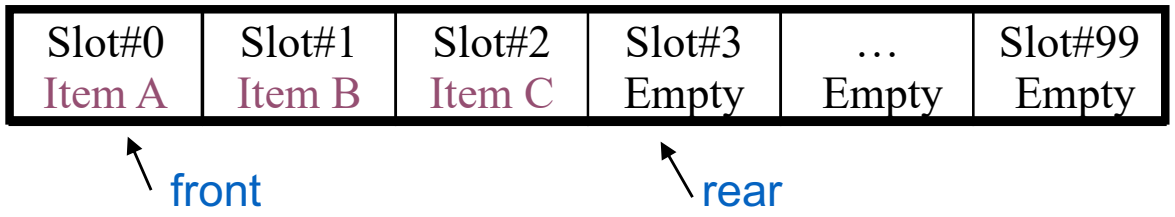



Suppose some items are appended into the queue:

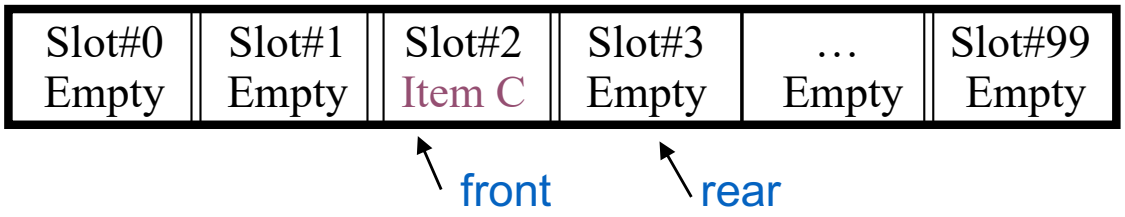


# Implementation of Queue Sequential Allocation (Using Array)

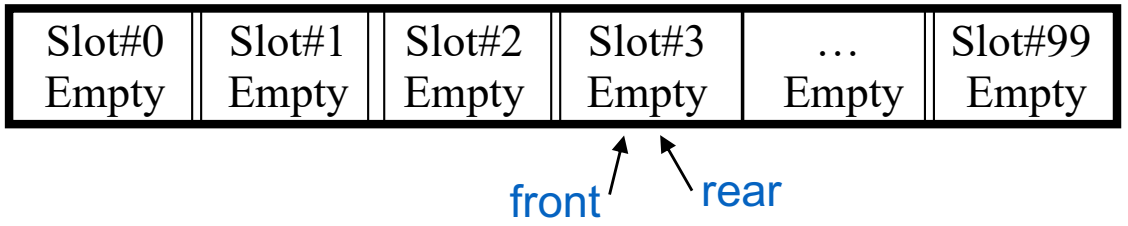
```
#define TOTAL_SLOTS 100
class MyQueue
{
    private:
        int front;
        int rear;
        int items[TOTAL_SLOTS];
};
```




Suppose we remove 2 items:

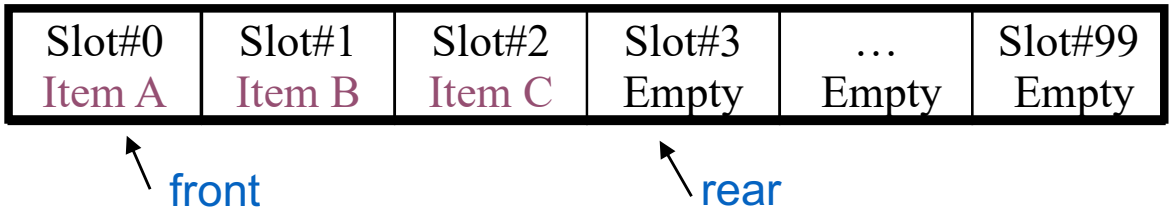



Then we remove the remaining one item:




**If the queue is <u>empty</u>, we'll have_____.**

```
#define TOTAL_SLOTS 100
class MyQueue
{
    private:
        int front;
        int rear;
        int items[TOTAL_SLOTS];
};
```




Suppose 2 items are removed and 96 items added:


Then, we add (append) item YY:


Then, we add (append) one more item ZZ:


**However, we can't add further item. Reason: we should not let rear = front if the queue is not empty. (The queue is empty when rear = front)**

**Hence, this implementation allows only "_____" items.**

# Implementation of Queue Sequential Allocation (Using Array)

```
#define TOTAL_SLOTS 100
class MyQueue
{
    private:
        int front;   //the index of the front slot that contains the front item
        int rear;    //the index of the first empty slot at the rear of queue
        int items[TOTAL_SLOTS];
    public:
        bool isEmpty();
        bool isFull();
        void enqueue(int );
        int dequeue();
};
```

# Implementation of Queue Sequential Allocation (Using Array)

```
bool MyQueue::isEmpty()
{
        return (front==rear);
}
bool MyQueue::isFull()
{
        return((rear+1)%TOTAL_SLOTS==front);
}
```

```
void MyQueue::enqueue(int data)
{
        if(!isFull())
        {
                items[rear]=data;
                rear=(rear+1) %TOTAL_SLOTS;
        }
}
```

```
int MyQueue::dequeue( )
{
        int ret_val;
        if(!isEmpty())
        {
                ret_val=items[front];
                front=(front+1)%TOTAL_SLOTS;
                return ret_val;
        }
}
```

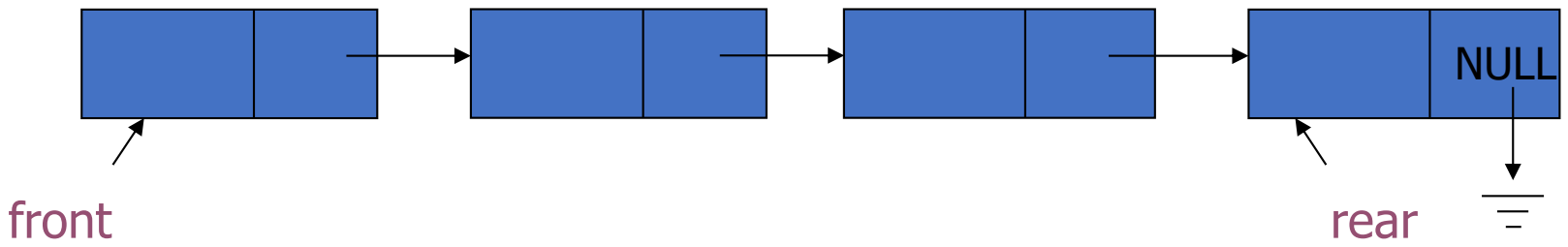
# Exercise 1

Suppose an intermixed sequence of queue enqueue and dequeue operations are performed. The enqueues enqueue the integers 0 through 9 in order; the dequeues print out the returned value.

Can the sequence *4 3 2 1 0 5 6 7 8 9* be printed as a result?

If the answer is yes, indicate the sequence of enqueues and dequeues needed. If the answer is no, explain why.

# Implementation of Queue Using Linked List




- Queue can also be implemented with linked list.
- A pointer **front** points to the first node of the queue.
- A pointer **rear** points to the last node of the queue.
- If the queue is **empty**, then **front=rear=NULL**.
- When will the queue be full?

# Linked Implementation of Queue

```
// MyQueue.h
#include “stdlib.h”
{
      class MyQueue
      {
            public:
                  MyQueue( );
                  bool IsEmpty();
                  void Enqueue(int );
                  int Dequeue();
            private:
                  ListNode* front;
                  ListNode* rear;
                  int size;
      };
}
```

```
// MyQueue.cpp

#include “MyQueue.h”
MyQueue::MyQueue()
{
            size=0;
            front=NULL;
            rear=NULL;
}
bool MyQueue::IsEmpty()
{
            return (front==NULL);
}
```

# Linked Implementation of Queue

**To insert an item (Enqueue)**

**We have 2 cases:**

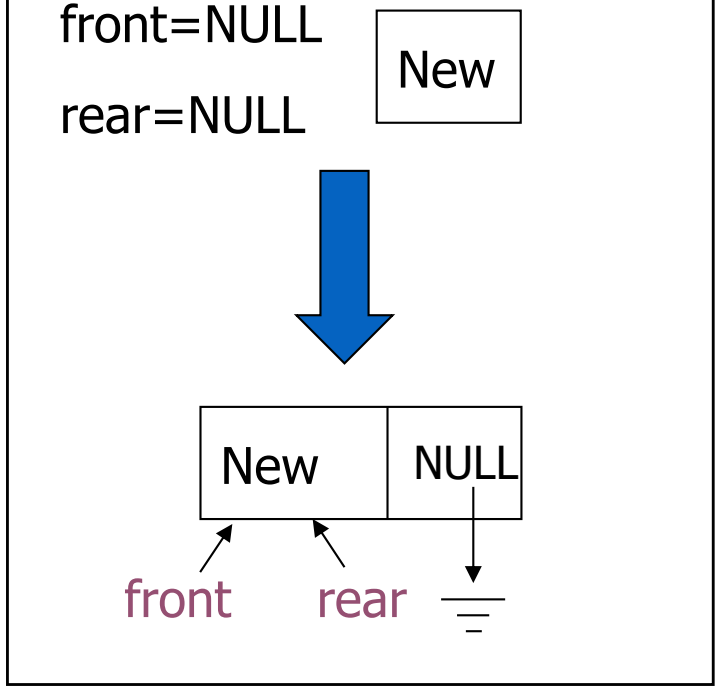
**The queue is empty or not.**

Step 1: Allocate a new slot, **p**, to store the item.

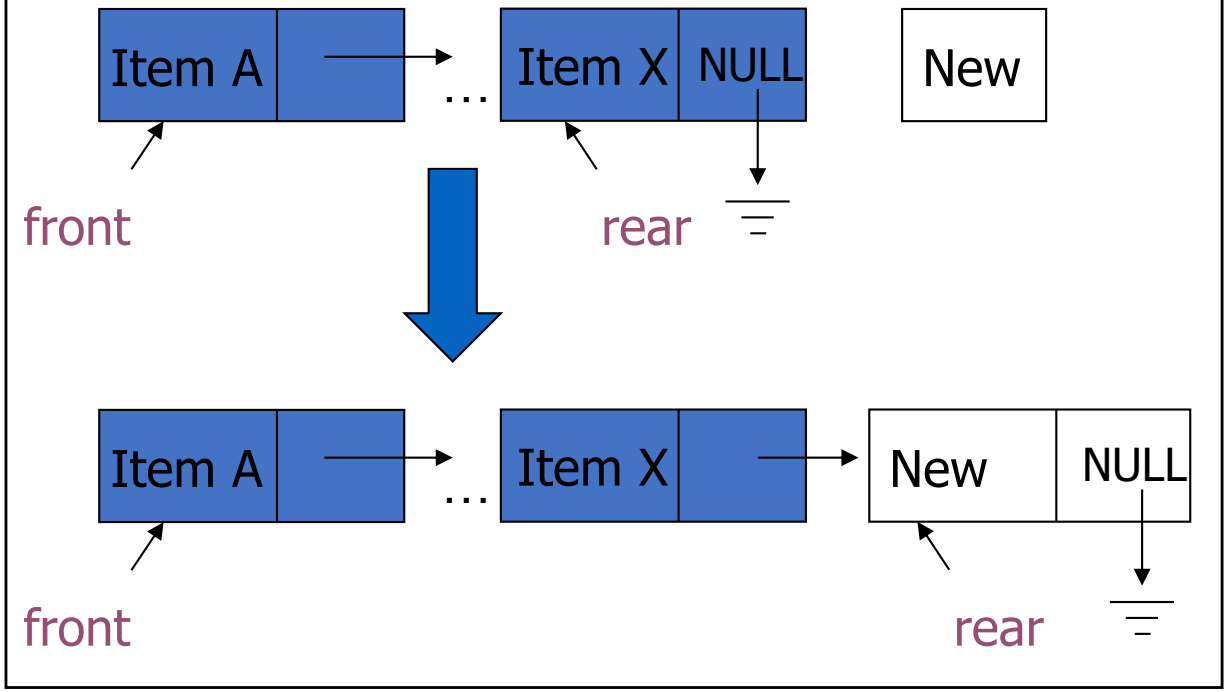
Step 2: Connect **p** to the queue (2 cases).

Step 3: Update the rear pointer to point to **p**.

Case 1: The queue is empty




Case 2: The queue is not empty



# Linked Implementation of Queue

**To insert an item (Enqueue)**

**We have 2 cases:**

**The queue is empty or not.**

Step 1: Allocate a new slot, **p**, to store the item.

Step 2: Connect **p** to the queue (2 cases).

Step 3: Update the pRear pointer to point to **p**.

```
// MyQueue.cpp

#include "MyQueue.h"
void MyQueue::Enqueue(int data)
{
        ListNode *p=new ListNode(data);
        if (IsEmpty())
                front=p;
        else
                rear->next=p;
        rear=p;
}
```

# Linked Implementation of Queue

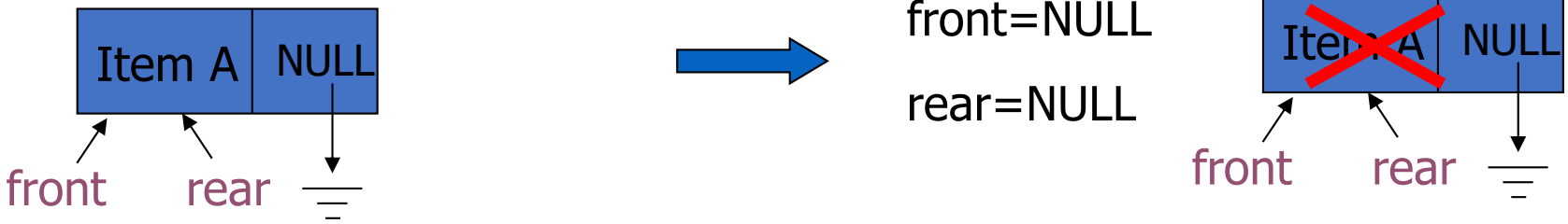
**To delete an item (the front item) and return it**

**We have 3 cases:**

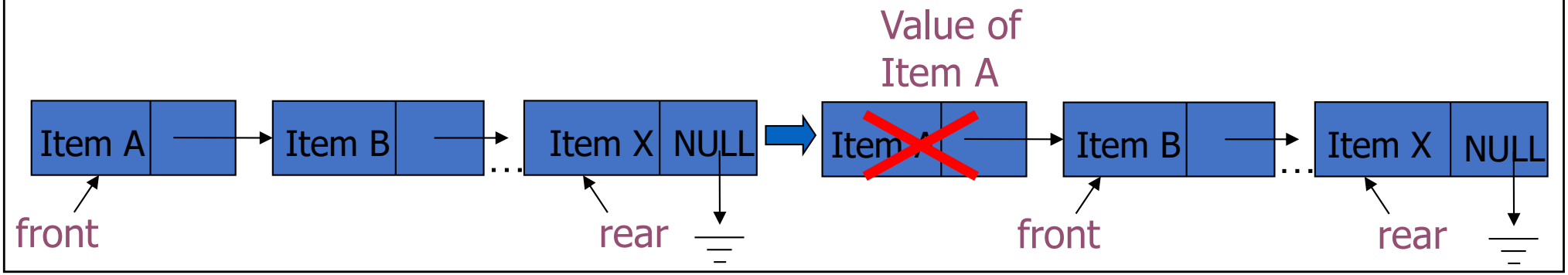
**The queue has 0 item, 1 item or more than one item.**

Case 1: The queue has 0 item → Output error

Case 2: The queue has 1 item




Case 3: The queue has more than one item



# Linked Implementation of Queue

**To delete an item (the front item) and return it**

**We have 3 cases:**

**The queue has 0 item, 1 item or more than one item.**

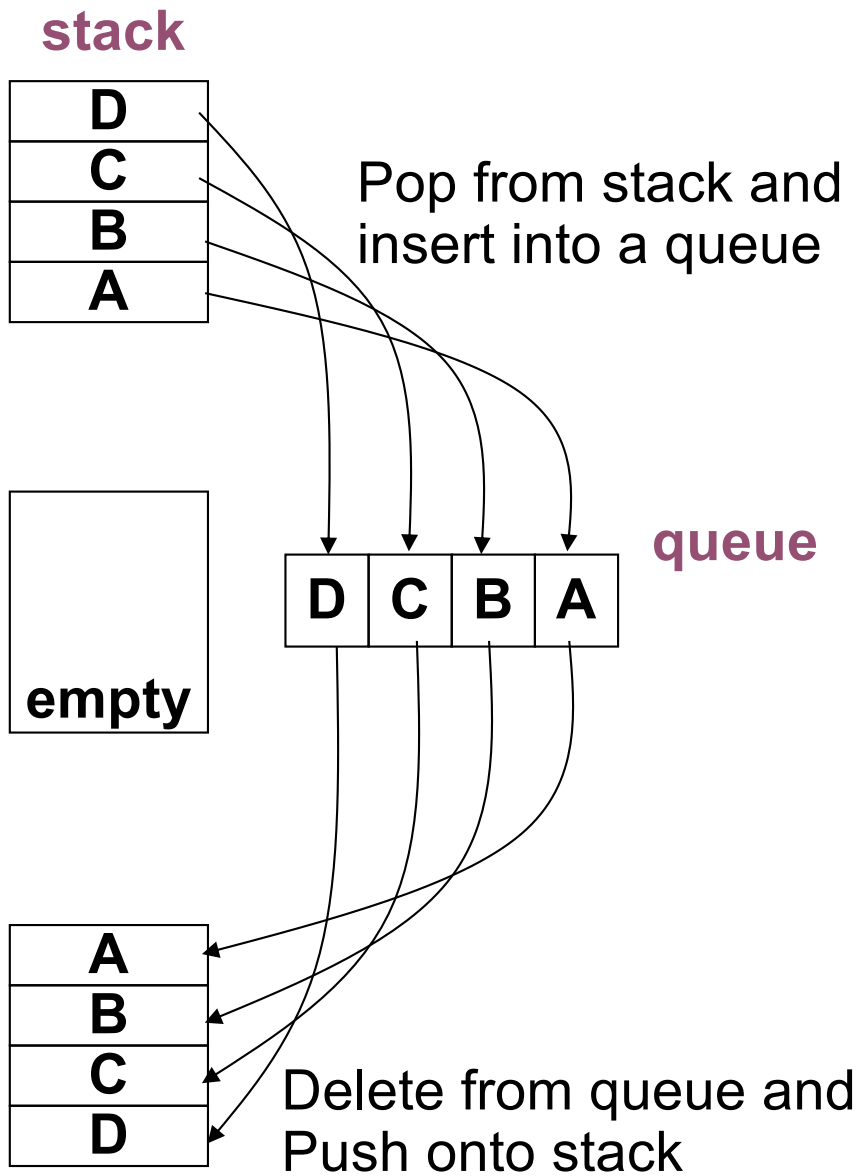
```
// MyQueue.cpp

#include "MyQueue.h"
int MyQueue::Dequeue()
{
        int ret_value;
        if (!IsEmpty())
        {
                ret_value=front->data;
                front=front->next;
                if(front==NULL)
                        rear=NULL;
        }
        return ret_value;
}
```

# Application 1: Reversing a Stack

**Reversing a stack**

```
Stack *s;
Queue *p;
…
while(!s->IsEmpty())
{
    x = s->pop();
    p->Enqueue(x);
}
while (!p->IsEmpty())
{   x = p->Dequeue();
    s->push(x);
}
```



# Application 2: Phenomena on the computer

- See online movies
- The way printer works
- Round Robin Schedule
  - Establish a queue for current jobs
  - Whenever a time slot is used up
    - Insert the current job into the queue
    - Begin executing the job fetched from the queue

# Round Robin Schedule

- job 1 : 4 time slots; job 2 : 3 time slots
- job 3 : 1 time slot; job 4 : 2 time slots

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1(4 left) | 2(3 left) | 3(1 left) | 4(2 left) | | | | |
| | 2(3 left) | 3(1 left) | 4(2 left) | 1(3 left) | | | |
| | | 3(1 left) | 4(2 left) | 1(3 left) | 2(2 left) | | |
| | | | 4(2 left) | 1(3 left) | 2(2 left) | | |
| | | | | 1(3 left) | 2(2 left) | 4(1 left) | |
| | | | | | 2(2 left) | 4(1 left) | 1(2 left) |
| 2(1left) | | | | | | 4(1 left) | 1(2 left) |
| 2(1left) | | | | | | | 1(2 left) |
| 2(1left) | 1(1 left) | | | | | | |
| | 1(1 left) | | | | | | |
| | | | | | | | |

# Exercise 1

Implement Queue using Stacks.

1) enQueue is O(N), deQueue is O(1)
2) enQueue is O(1), deQueue is O(N)

# Exercise 1

```
struct Queue {
    stack<int> s1, s2;
    void enQueue(int x)
    {
        // Move all elements from s1 to s2
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }
        // Push item into s1
        s1.push(x);
        // Push everything back to s1
        while (!s2.empty()) {
            s1.push(s2.top());
            s2.pop();
        }
    }
    int deQueue()
    {
        // if first stack is empty
        if (s1.empty()) {
            return -1;
        }

        // Return top of s1
        int x = s1.top();
        s1.pop();
        return x;
    }
}
```

```
struct Queue {
    stack<int> s1, s2;

    // Enqueue an item to the queue
    void enQueue(int x)
    {
        s1.push(x);
    }

    // Dequeue an item from the queue
    int deQueue()
    {
        // if both stacks are empty
        if (s1.empty() && s2.empty()) {
            return -1;
        }
        // if s2 is empty, move
        // elements from s1
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        // return the top item from s2
        int x = s2.top();
        s2.pop();
        return x;
    }
};
```

# Queue enough?

- In Game, when factory produce units
  - Suppose a factory can produce the following three units:
    - Attacker
    - Defender
    - Worker
  - When you are giving commands, you do not have so much time to worry about the order of production. It should be AI's work to arrange that for you

Least important

Most important

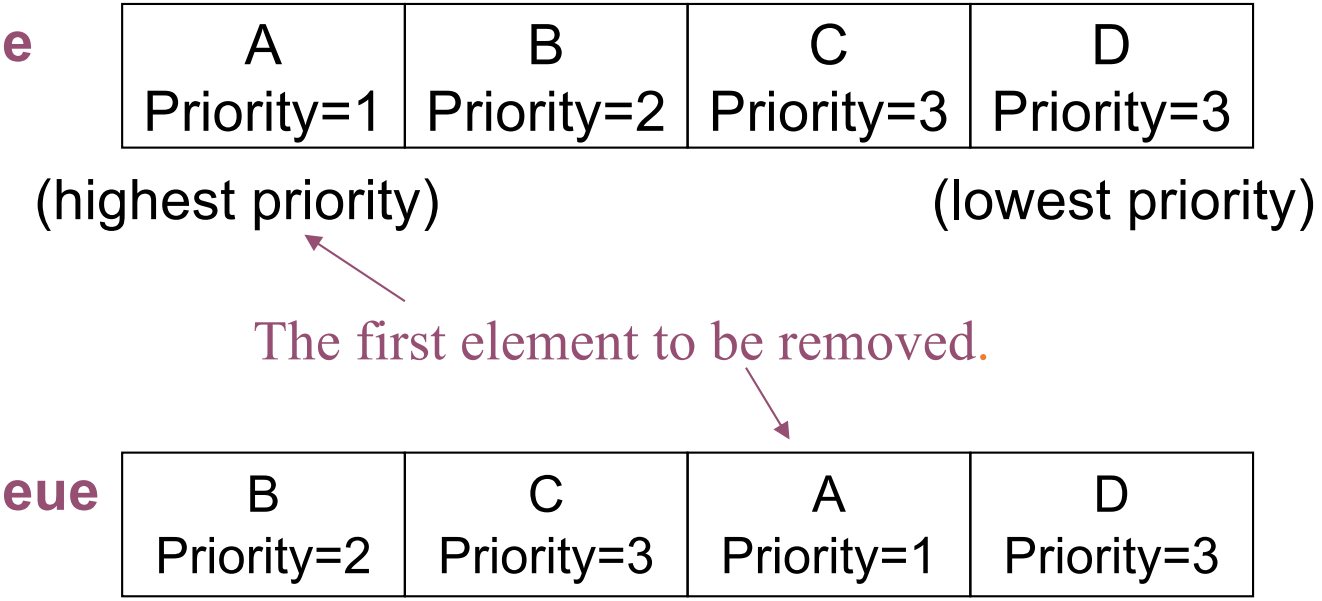
Moderately important

# Priority Queue

## Priority Queue

- The elements in a stack or a FIFO queue are ordered based on the sequence in which they have been inserted.
- In a priority queue, the sequence in which elements are removed is based on the priority of the elements.

**Ordered Priority Queue**




**Unordered Priority Queue**

| B<br>Priority=2 | C<br>Priority=3 | A<br>Priority=1 | D<br>Priority=3 |
|---|---|---|---|

# Priority Queue

## Priority Queue - List Implementation

- To implement a priority queue as an **ordered** list.

Time complexity of the operations :
(assume the sorting order is from highest priority to lowest)

**Insertion**: Find the location of insertion. *O(n)*
Link the element at the found location. *O(1)*
Altogether: *O(n)*

**Deletion**: The highest priority element is at the front.
i.e., Remove the front element takes ***O(1)*** time

# Priority Queue

## Priority Queue - List Implementation

- To implement a priority queue as an **unordered** list.

Time complexity of the operations :

**Insertion**: Simply insert the item at the rear. *O(1)*

**Deletion**: Traverse the entire list to find the maximum priority element. *O(n)*.

Copy the value of the element to return it later. *O(1)*

Delete the node. *O(1)*

Altogether: *O(n)*

We will come back to this after we learned trees

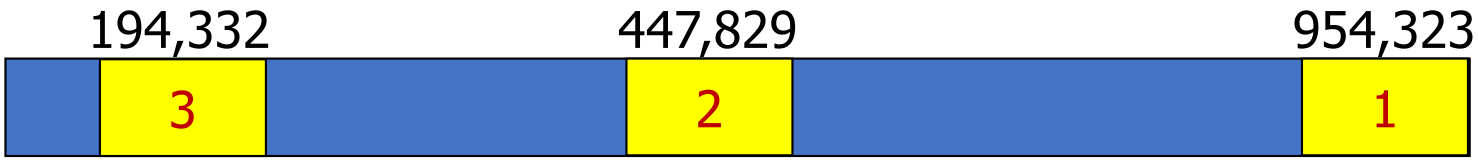
# Sparse data

- There are many players in a complex game
- Each player has an identification number (key)
- The range of the keys can be 1~1,000,000
- No two players have the same key
- Suppose now we have 3 players
  - 954,323
  - 447,829
  - 194,332
- They are far away from each other
- The player information is called key-based data

# Sparse data

- How to store those data in the computer so that we can easily get the player's information by their keys?
  - Array:

    A lot of memory space wasted




  - Linked List:

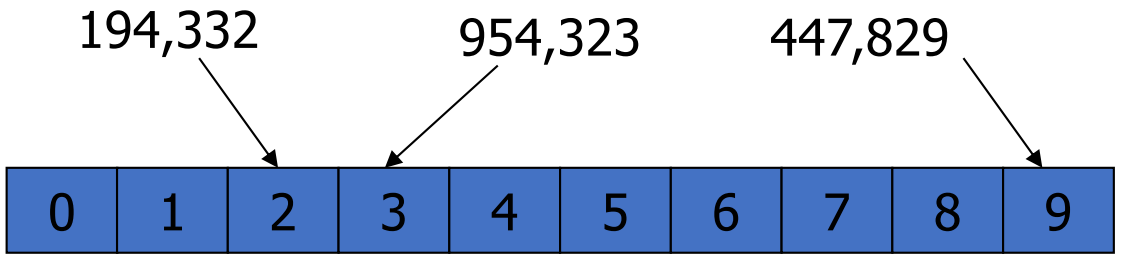
    Hard to search if we have 10,000 players

  - Hash Table

    Best solution in this case!

# Basic Hash Table

- Advantages:
  - Quickly store sparse key-based data in a reasonable amount of space
  - Quickly determine if a certain key is within the table

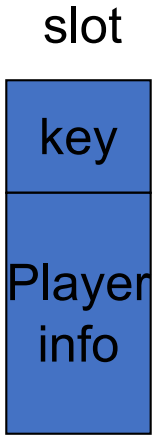



194,332%10=2 or 194,332≡2 (mod 10)

447,879%10=9 or 447,879≡9 (mod 10)

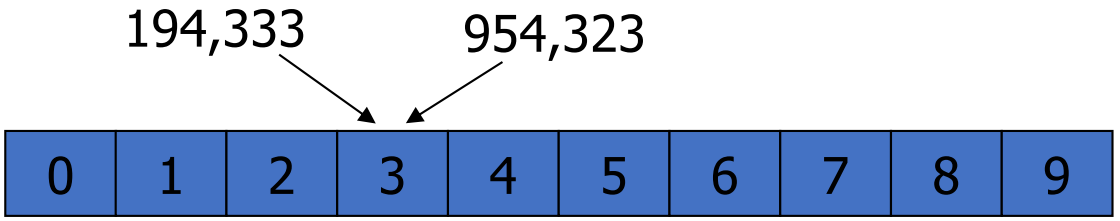
954,323%10=3 or 954,323≡3 (mod 10)

To get the information, we use:
player=table[key%10];



# Collisions

- Two players mapped to the same cell




- Method to deal with collisions
  - Change the table
  - Hash functions
    - ‘Hash’ in the dictionary: chop (meat) into small pieces
    - Here, we ‘Hash’ numbers

# Hash Functions

**Good hash function:**

Fast computation, Minimize collision

**Kinds of hash functions:**

- **Division**: Slot_id = Key % table_size.
- **Others**: eg., Slot_id = ($Key^2$ + Key + 41) % table_size
- **table_size should better be a prime number.**

# Hash Functions

We have the following keys: 12, 22, 32, 42, 52, 62, 72, 82.

Using a hash table size of 10, we calculate the slot IDs for each key :
Key: 12, Slot_id = 12 % 10 = 2
Key: 22, Slot_id = 22 % 10 = 2 (Collision with previous key)
Key: 32, Slot_id = 32 % 10 = 2 (Collision with previous keys)
Key: 42, Slot_id = 42 % 10 = 2 (Collision with previous keys)
Key: 52, Slot_id = 52 % 10 = 2 (Collision with previous keys)
Key: 62, Slot_id = 62 % 10 = 2 (Collision with previous keys)
Key: 72, Slot_id = 72 % 10 = 2 (Collision with previous keys)
Key: 82, Slot_id = 82 % 10 = 2 (Collision with previous keys)

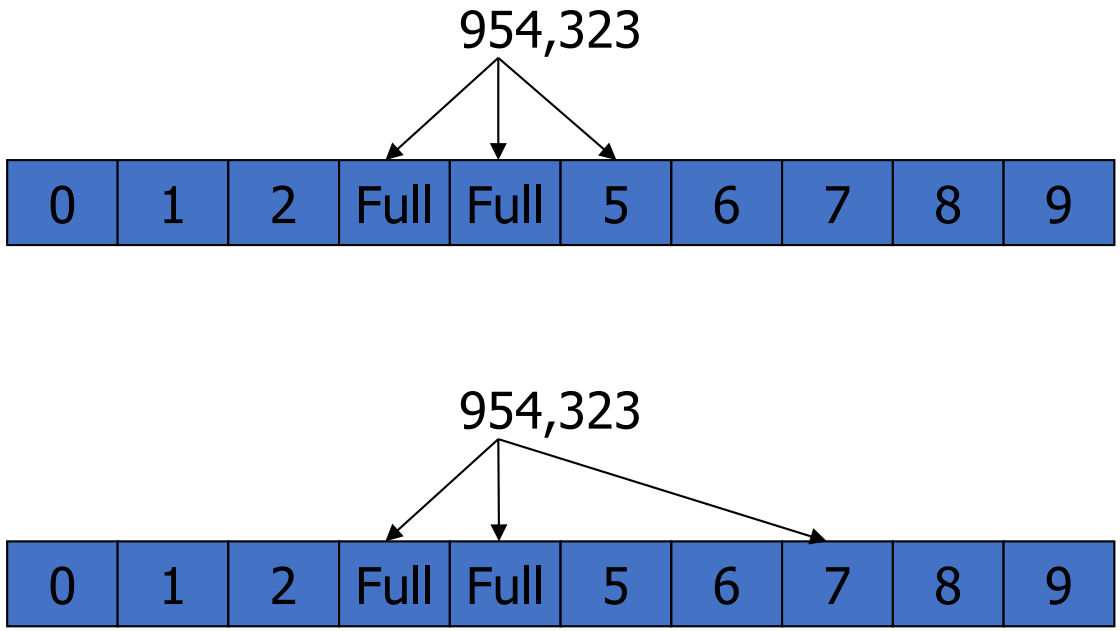
Using a table size of 11 (a prime number):
Key: 12, Slot_id = 12 % 11 = 1
Key: 22, Slot_id = 22 % 11 = 0
Key: 32, Slot_id = 32 % 11 = 10
Key: 42, Slot_id = 42 % 11 = 9
Key: 52, Slot_id = 52 % 11 = 8
Key: 62, Slot_id = 62 % 11 = 7
Key: 72, Slot_id = 72 % 11 = 6
Key: 82, Slot_id = 82 % 11 = 4

# Combination of Hash Functions

- Collision is easy to happen if we use ***%*** function
- Combination:
  - Apply hash function $h_1$ on key to obtain *mid_key*
  - Apply hash function $h_2$ on *mid_key* to obtain *Slot_id*
- Example:
  - We apply %101 on 12320324111220 and get 79
  - We apply %10 on the result 79 obtained by %101
    - 79 % 10 =9

# Collision Resolution - Open Addressing

- Linear Probing
  - If collide, try Slot_id+1, Slot_id+2
- Quadratic Probing
  - If collide, try Slot_id+1, Slot_id+4,…

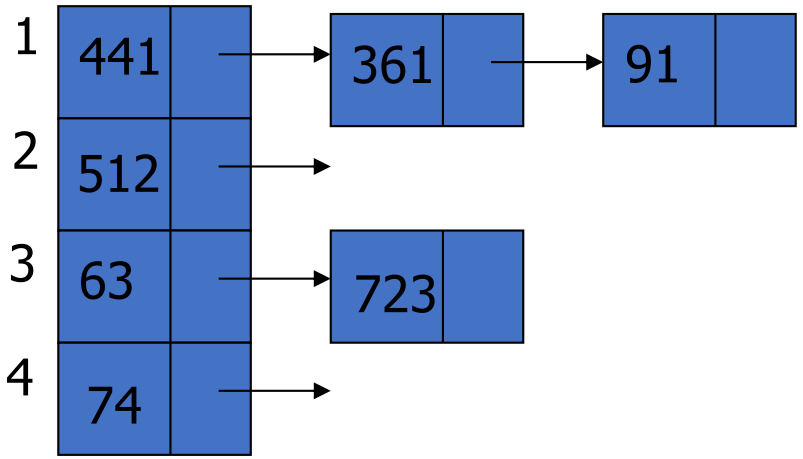


# Collision Resolution - Open Addressing

- Linear Probing
  - If collide, try Slot_id+1, Slot_id+2
- Quadratic Probing
  - If collide, try Slot_id+1, Slot_id+4,…
- Double Hashing
  - If collide, try Slot_id+$h_2(x)$, Slot_id+$2h_2(x)$,… (prime size important)
- General rule: If collide, try other slots in a certain order
- How to find data?
  - If not found, try the next position according to different probing rule
  - Every key has a preference over all the positions
  - When finding them, just search in the order of their preferences

# Collision Resolution - Separate Chaining

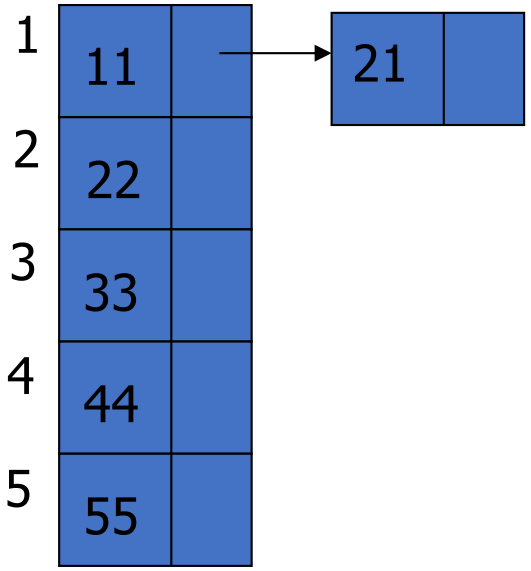
- Problems with Open Addressing?
- Using linked list to solve Collision
  - Every slot in the hash table is a linked list
  - Collision→Insert into the corresponding list
  - Find data→Search the corresponding list



# Collision Resolution

- Example: 11,22,33,44,55,66,77,88,99,21
  - Using linear probing


  - Using separate chaining



# More on Hash Table Size

Table of prime size is important in the following case:

For quadratic probing, we have the following property:

> If quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty.

# Rehashing

- Too many elements in the table
  →Too many collisions when inserting
- **Load factor** = number of slots occupied/total slots
- When half full, rehash all the elements into a double-size table
- In interactive systems, the user who triggers rehashing is unlucky
- In total, only O(n) cost incurred for a hash table of size n

- Example: initial hash table size 2, when the size grows to 32, how many rehashes are done?
  - 2→4 1 number rehashed
  - 4→8 2 numbers rehashed
  - 8→16 4 numbers rehashed
  - 16→32 8 numbers rehashed
  - In total, 15 numbers rehashed, 15<16=32/2

# More on Rehashing

- How can rehashing be used?
  - If we allow rehashing, then quadratic probing can always succeed in inserting new items because the table will always be at least half empty.
- How to keep the table size still prime when you do rehashing?

# Application —— Dictionary

- How do Word perform spelling check?
- A dictionary (large hash table) is kept
- Hash words into that dictionary
- The way to hash words
  - Establish a map between characters and numbers
  - E.g. A—136, F—356, T—927, E—442, R—091
  - “AFTER” corresponds to the key 136,356,927,442,091
  - Hashing ‘AFTER’ will be equivalent to hashing the key

# How to write Hash Class?

- Exercises:
  - ➢1. use linear probing to write a hash class (array based)
  - ➢2. use this class to implement your own dictionary (ASCII value for each character)

```
class HashTable {
private:
    const int TABLE_SIZE = 100;  // Size of the hash table

    ... // other private member variables

    int hashFunction(const string& word) {
        ... // implementation
    }

public:
    void insert(const std::string& word) {
        ... // implementation
    }

    bool search(const std::string& word) {
        ... // implementation
    }
};
```

```
int main() {
    HashTable dictionary;

    // Insert some words into the dictionary
    dictionary.insert("AFTER");
    dictionary.insert("FAR");

    // Search for words in the dictionary
    cout << "AFTER: " <<
(dictionary.search("AFTER") ? "Found" :
"Not Found") << endl;

    cout << "RAT: " <<
(dictionary.search("RAT") ? "Found" : "Not
Found") << endl;
    return 0;
}
```

```
private:
    static const int TABLE_SIZE = 100;
    std::vector<std::string> table[TABLE_SIZE];

    int hashFunction(const std::string& word) {
        int sum = 0;
        for (char c : word) {
            sum += c;
        }
        return sum % TABLE_SIZE;
    }
```

```
public:
    void insert(const std::string& word) {
        int index = hashFunction(word);

        // Handle collisions with linear probing
        while (!table[index].empty())
            index = (index + 1) % TABLE_SIZE;

        table[index].push_back(word);
    }

    bool search(const std::string& word) {
        int index = hashFunction(word);

        // Handle collisions with linear probing
        while (!table[index].empty()) {
            if (table[index][0] == word)
                return true;

            index = (index + 1) % TABLE_SIZE;
        }

        return false;
    }
```

# Learning Objectives

1. Explain the concepts of Queue
2. Understand the two functions of Queue
3. Able to solve simple applications using Queue
4. Understand the concept of Priority Queue and its two implementations

D:1;  C:1,2;  B:1,2,3;  A:1,2,3,4

# Learning Objectives

1. Understand the concept of Hash
2. Able to insert step by step in a hash table given the data and the probing rule
3. Know the property of Quadratic Probing and Double Hashing
4. Able to Implement Hash Table

D:1;  C:1,2;  B:1,2,3;  A:1,2,3,4

# Exercise 1

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \pmod{()}\ 10$, show the resulting

a. separate chaining hash table
b. hash table using linear probing
c. hash table using quadratic probing
d. hash table with second hash function $h2(x) = 7 - (x \bmod 7)$