Data Structures & Algorithms Design- SS ZG519
Lecture - 3

BITS Pilani
Pilani Campus

Dr. Padma Murali

Lecture 3 Topics

- Stack & Queue;
- Analysis of Algorithms -- space and time complexity

- Slides source: 2008 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
- Lecture notes

Running Time (Example)

```
sum = 0;
{
for (k=1; k<=n; k++)
   for (j=1; j<=k; j++)
      sum++;
}
```

What is the running time for this code?

Running Time (Example)

Number of executions

k	1	2	3	….	n
j	1	1,2	1,2,3	…	1,2,. n
# runs	1	2	3	…	n

Running Time (Example)

$$\# \text{ runs} = 1 + 2 + 3 + 4 \ldots + n = \sum_{j=1}^{n} j$$

$$\sum_{j=1}^{n} j = \quad n(n+1)/2 \qquad = n^2$$

$T(n) = c1 + c2\ (n+1) + c3(n^2 + 1) + c4\ (n^2\ ) =$ Order of n^2

Running Time (Example)

What is the running time for the following codes?

a)
```
 sum1 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=n; j++)
        sum1++;
```

b)
```
 sum2 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=k; j++)
        sum2++;
```

c)
```
sum3 = 0;
for (k=1; k<=n; k++)
  for (j=1; j<=n; j++)
        sum3++;
```

Running Time (Example a)

innovate achieve lead

Number of executions

k	1	2	4	….	n
j	1,2,..n	1,2,..n	1,2..n	…	1,2,. n
# runs	n	n	n	…	log n

N x log N

Running Time (Example a)

$$\# \text{ runs} = (1 + ..N) \log n = \sum_{j=1}^{\log n} n$$

$$\sum_{j=1}^{\log n} n = n \log n$$

T(n) = Order of $n \log n$.

Running Time (Example b)

Number of executions

k	1	2	4	….	n
j	1	1,2	1,2,3,4	…	1,2,. n
# runs	1	2	4	…	log n

1 + 2 + 4 + 8 + 16 + …… n

Running Time (Example b)

$\#$ runs = 1+2+4+8+16 ..+ n

$= 1 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots\ldots + 2^{\log n}$

$$\sum_{j=1}^{\log n} 2^i = 2n-1$$

T(n) = Order of n.

Growth Rate

The growth rate for an algorithm is the rate at which the *cost of the algorithm grows as the size of the input grows.*

- *Linear Growth $T(n) = n$*
- *Quadratic Growth $T(n) = n^2$*
- *Exponential Growth $T(n) = 2^n$*
- *Logarithmic Growth $T(n) = n \log n$*

Types of Analysis

Not all inputs of a given size take the same time to run.

- Best case
- Worst case
- Average case

Types of Analysis

- The *best case running time* of an algorithm is the function defined by the minimum number of steps taken on any instance of size n.
- The *worst case running time* of an algorithm is the function defined by the maximum number of steps taken on any instance of size n.
- The *average-case running time* of an algorithm is the function defined by an average number of steps taken on any instance of size n.

Types of Analysis

Worst case (e.g. numbers reversely ordered)

- Provides an upper bound on running time
- An absolute guarantee that the algorithm would not run longer, no matter what the inputs are

Best case (e.g., numbers already ordered)

- Input is the one for which the algorithm runs the fastest

Average case (general case)

- Provides a prediction about the running time
- Assumes that the input is random

Asymptotic Notations

A way to describe behavior of functions in the limit

- How we indicate running times of algorithms
- Describe the running time of an algorithm as n grows to ∞

O notation: asymptotic "less than": $f(n)$ "≤" $g(n)$

Ω notation: asymptotic "greater than": $f(n)$ "≥" $g(n)$

Θ notation: asymptotic "equality": $f(n)$ "=" $g(n)$

Asymptotic Notations - Examples

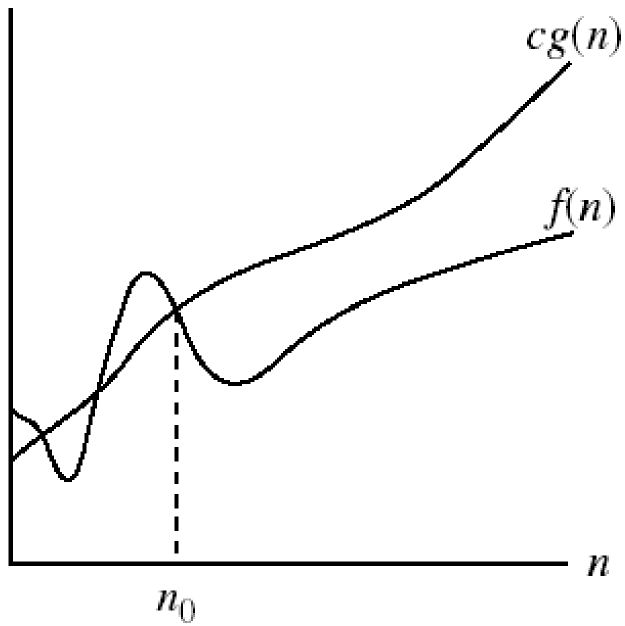
For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) is Θ(g(n)). Determine which relationship is correct.

- $f(n) = \log n^2$; $g(n) = \log n + 5$ $\quad f(n) = \Theta(g(n))$
- $f(n) = n$; $g(n) = \log n^2$ $\quad f(n) = \Omega(g(n))$
- $f(n) = \log \log n$; $g(n) = \log n$ $\quad f(n) = O(g(n))$
- $f(n) = n$; $g(n) = \log^2 n$ $\quad f(n) = \Omega(g(n))$

Asymptotic notations

O-notation

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



- Intuitively: $O(g(n))$ = the set of functions with a smaller or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Examples

$3n + 2 = O(n)$; $3n + 2 <= 4n$ for all $n >= 2$

$3n + 3 = O(n)$; $3n + 3 <= 4n$ for all $n >= 3$

$100n + 6 = O(n)$; $100n + 6 <= 101n$ for all $n >= 6$

Examples

$3n + 2 = O(n)$; $3n + 2 <= 4n$ for all $n >= 2$

$3n + 3 = O(n)$; $3n + 3 <= 4n$ for all $n >= 3$

$100n + 6 = O(n)$; $100n + 6 <= 101n$ for all $n >= 6$

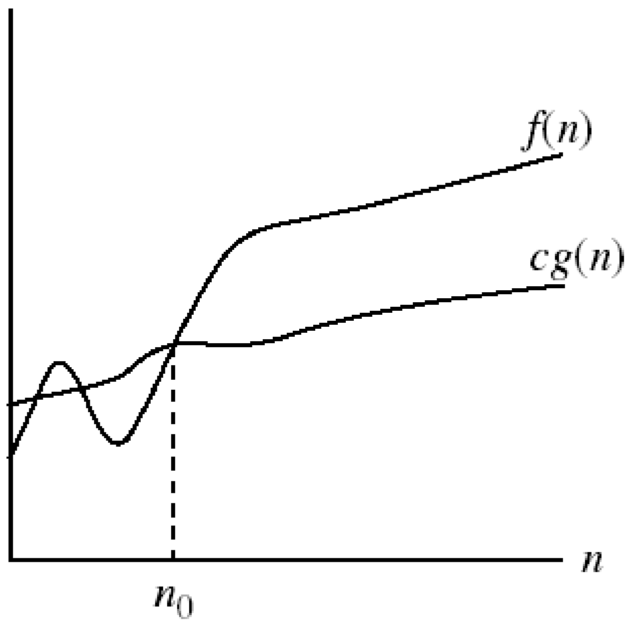
$10 n^2 + 4n + 2 = O(n^2)$

$10 n^2 + 4n + 2 <= 11 n^2$ for $n >= 5$

Asymptotic notations (cont.)

Ω - *notation*

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



- Intuitively: $\Omega(g(n))$ = the set of functions with a larger or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Examples

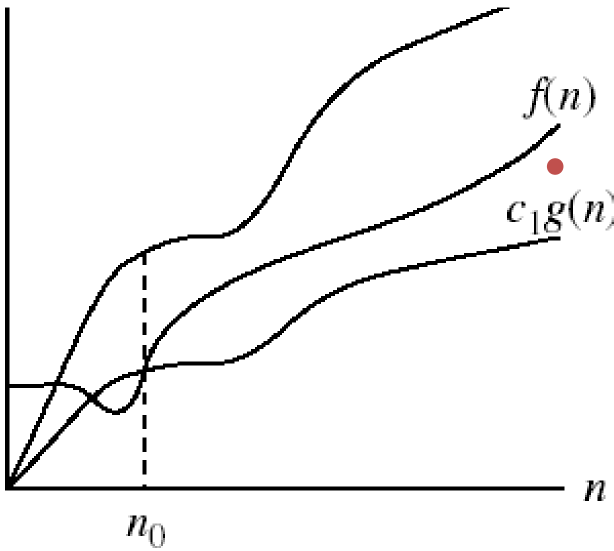
$3n + 2 = ?$ $3n + 2 >= 3n$ for all $n >= 1$

$3n + 3 = ?$ $3n + 3 >= 3n$ for all $n >= 1$

$100n + 6 = ?$ $100n + 6 >= 100n$ for all $n >= 1$

Asymptotic notations (cont.)

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



- Intuitively $\Theta(g(n))$ = the set of functions with the same order of growth as $g(n)$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Asymptotic Notations

A way to describe behavior of functions in the limit

- How we indicate running times of algorithms
- Describe the running time of an algorithm as n grows to ∞

O notation: asymptotic "less than": f(n) "≤" g(n)

Ω notation: asymptotic "greater than": f(n) "≥" g(n)

Θ notation: asymptotic "equality": f(n) "=" g(n)

Analysis of algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - theoretical analysis
  - empirical analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes most towards the running time of the algorithm

input size

running time

execution time for basic operation

Number of times basic operation is executed

$$T(n) \approx c_{op} C(n)$$

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	Number of list's items, i.e. *n*	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer *n*	*n*'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions
- Analyze the empirical data

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n
- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Types of formulas for basic operation's count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating order of growth with unknown multiplicative constant

  e.g., $C(n) \approx cn^2$

Order of growth

- Most important: Order of growth within a constant multiple as $n \rightarrow \infty$

- Example:
  - How much faster will algorithm run on computer that is twice as fast?

  - How much longer does it take to solve problem of double input size?

Values of some important functions as $n \to \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

Big-oh

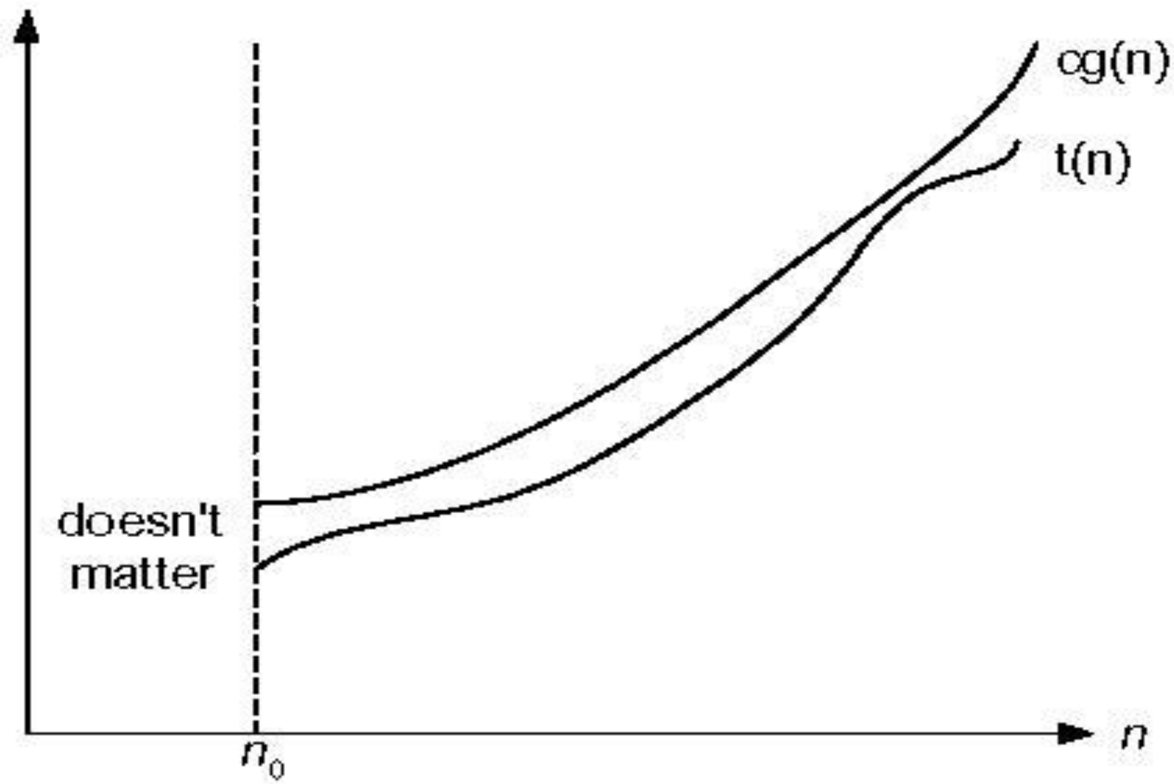


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

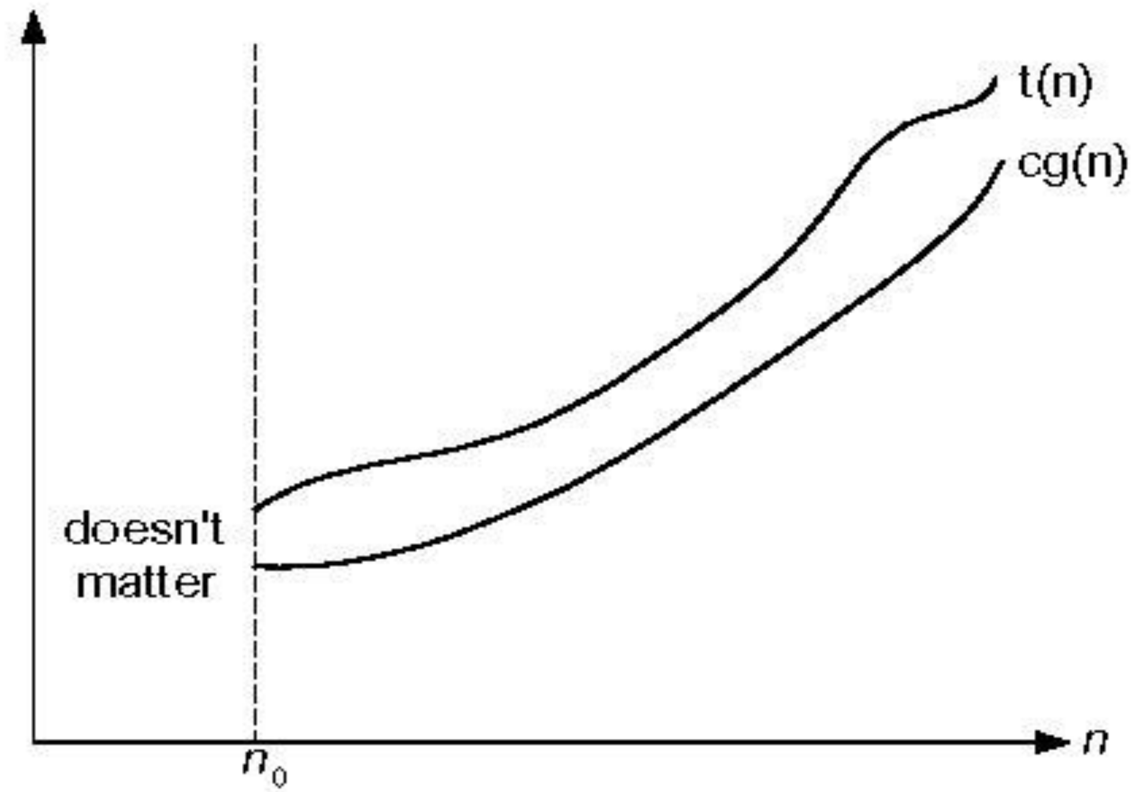


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

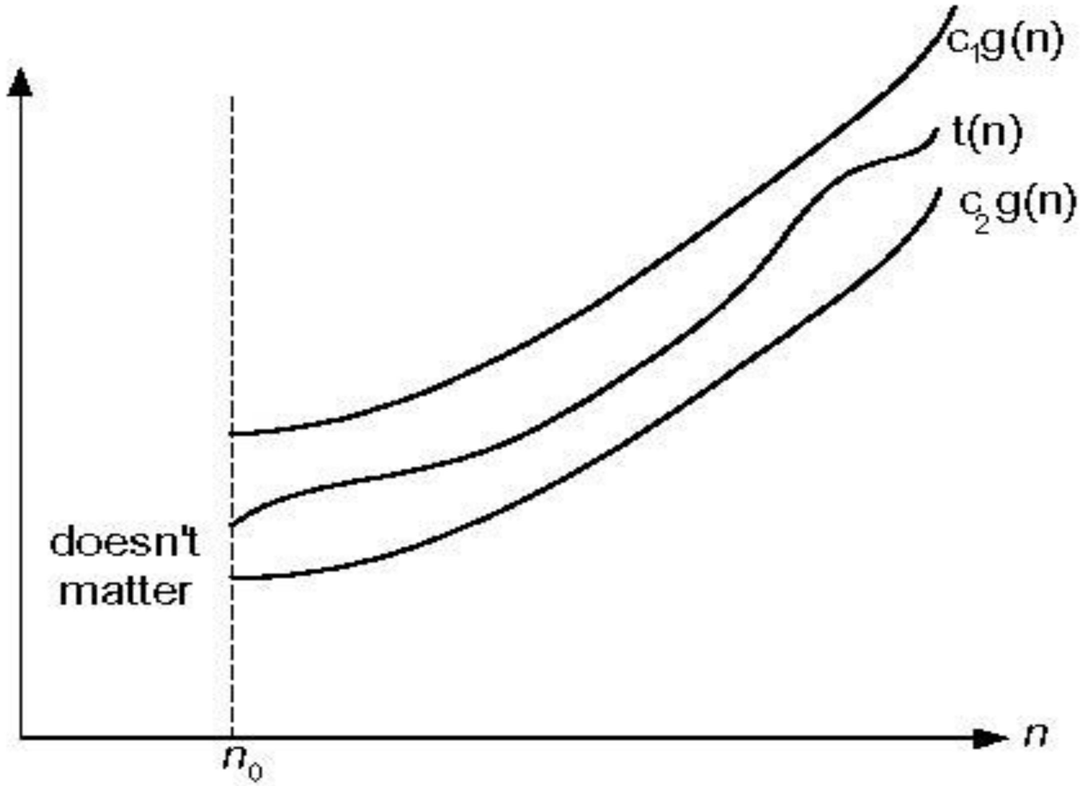
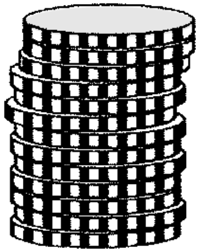


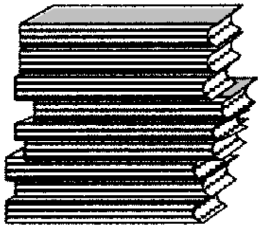
Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Stacks

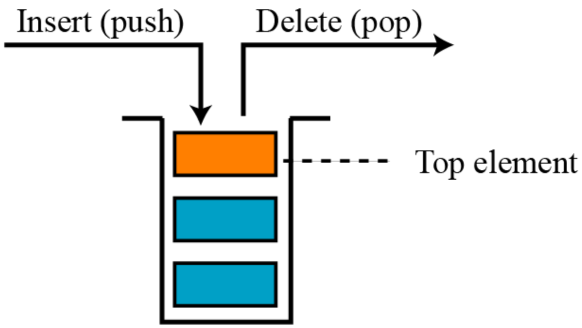
- A stack is a restricted linear list in which all additions and deletions are made at one end, the top. **(LIFO)**



Stack of coins



Stack of books

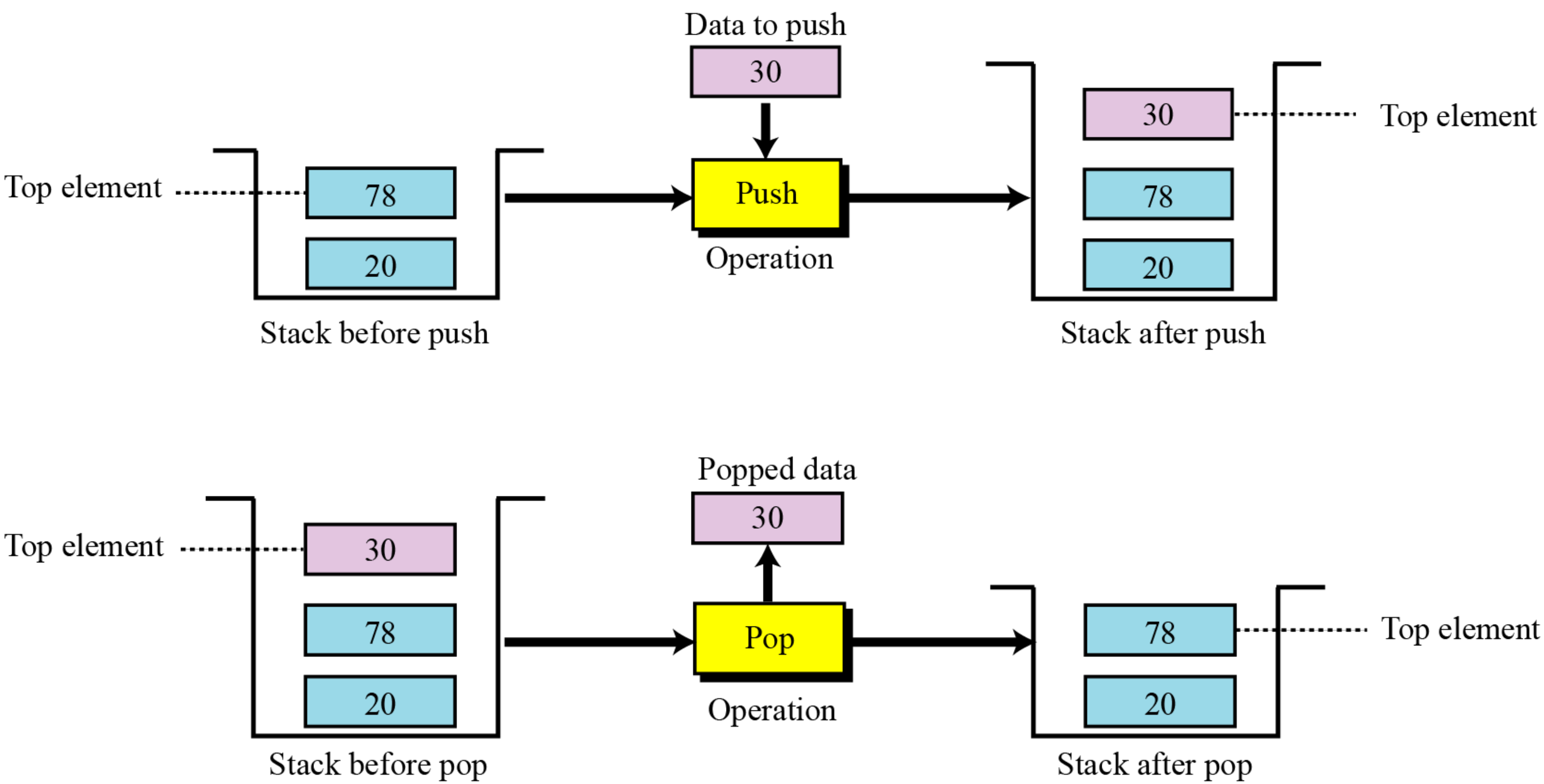


Computer stack

Operations on stacks

- **Stack ---** **stack** (stackName)
- **Push ---** **push** (stackName, dataItem)
- **Pop ----** **pop** (stackName, dataItem)
- **Empty---** **empty** (stackName)

Operations on stacks



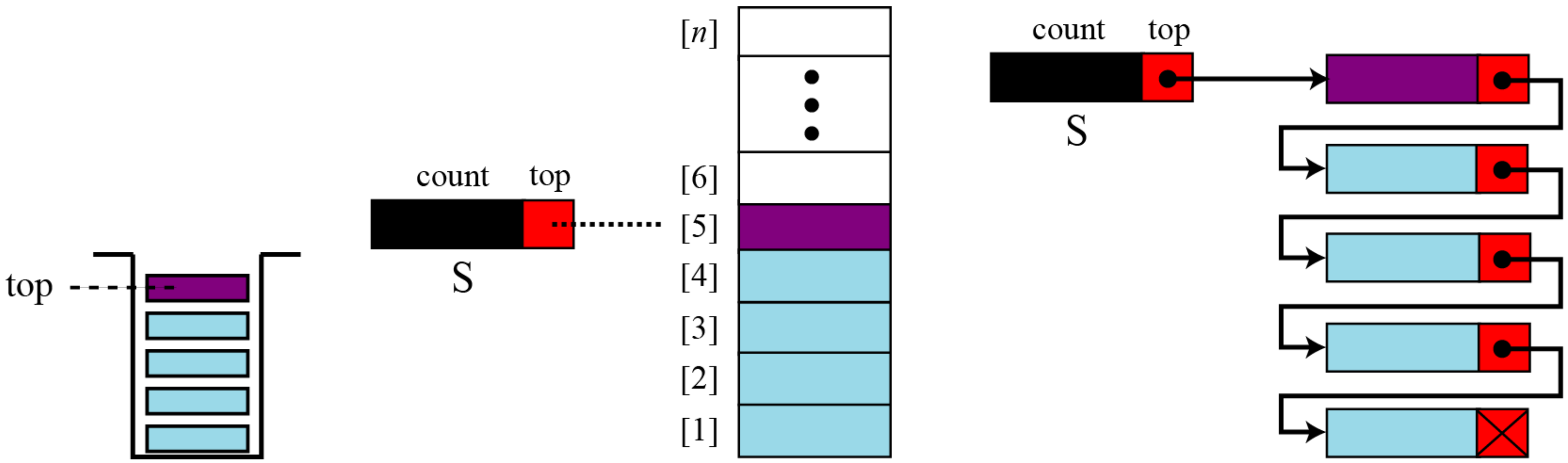
Stack ADT

Stack ADT

Definition	A list of data items that can only be accessed at one end, called the *top* of the stack.
Operations	**stack:** Creates an empty stack.
	push: Inserts an element at the top.
	pop: Deletes the top element.
	empty: Checks the status of the stack.

Stack implementation

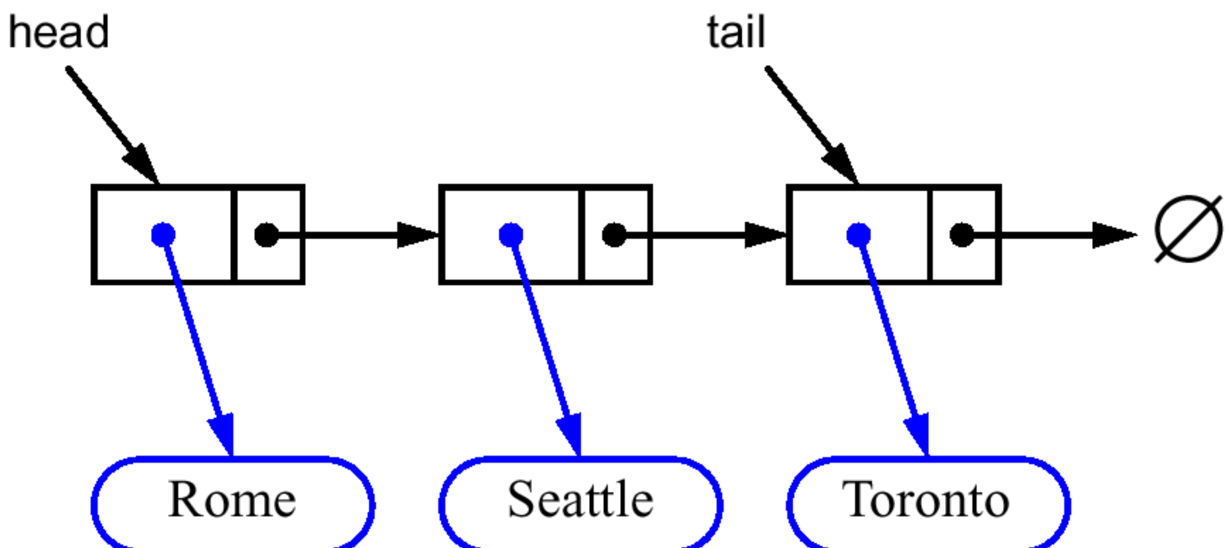
- Stack ADTs can be implemented using either an array or a linked list.



a. ADT b. Array implementation c. Linked list implemenation

Stacks: Singly Linked List implementation

- Nodes *(data, pointer)* connected in a chain by links



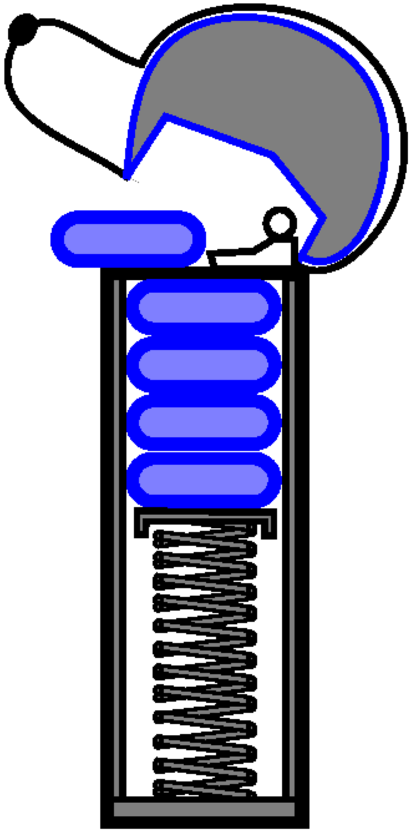
- the head or the tail of the list could serve as the top of the stack

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.

Stacks

- A coin dispenser as an analogy:



Stacks: An Array Implementation

- Create a stack using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

Stacks: An Array Implementation

- **Pseudo code**

```
Algorithm size()
return t+1

Algorithm isEmpty()
return (t<0)

Algorithm top()
if isEmpty() then
    return Error
return S[t]
```

```
Algorithm push(o)
if size()==N then
    return Error
t=t+1
S[t]=o

Algorithm pop()
 if isEmpty() then
    return Error
 t=t-1
return S[t+1]
```

Stacks: An Array Implementation

The array implementation is simple and efficient (methods performed in O(1)).

Disadvantage

There is an upper bound, *N*, on the size of the stack.

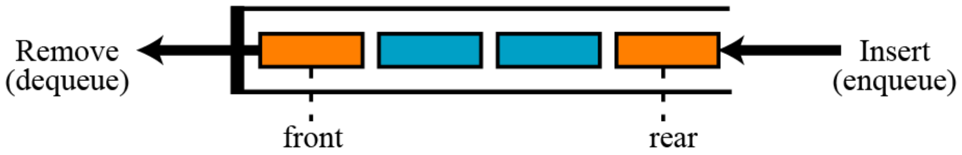
The arbitrary value *N* may be too small for a given application OR

a waste of memory.

Queues

- A queue is a linear list in which data can only be inserted at one end, called the *rear*, and deleted from the other end, called the *front.(FIFO).*

A queue of people



A computer queue

Operations on queues

- Queue
- Enqueue
- Dequeue
- Empty

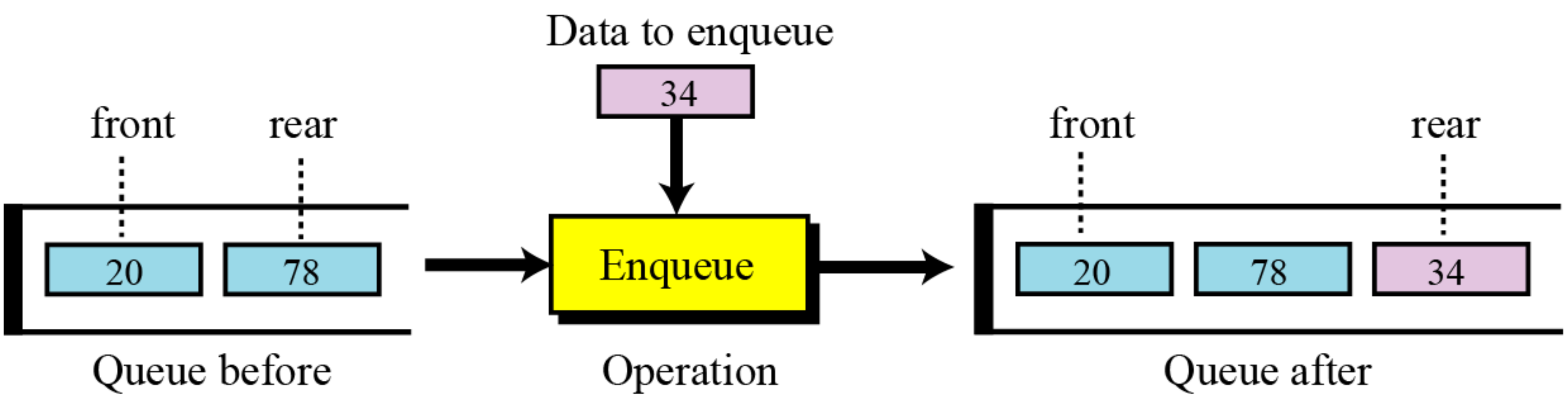
queue (queueName)

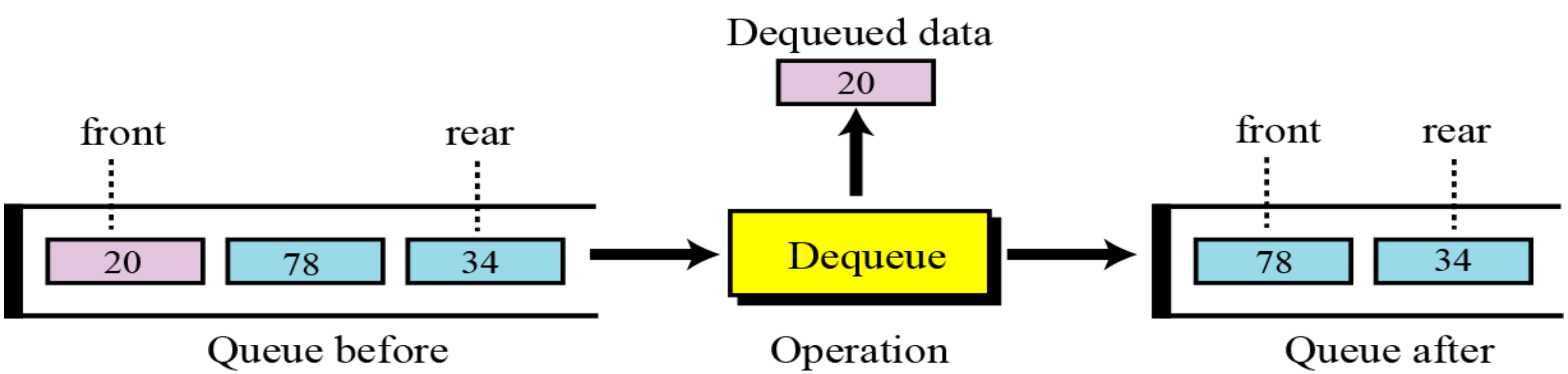
enqueue (queueName, dataItem)

dequeue (queueName, dataItem)

empty (queueName)

Operations on queues





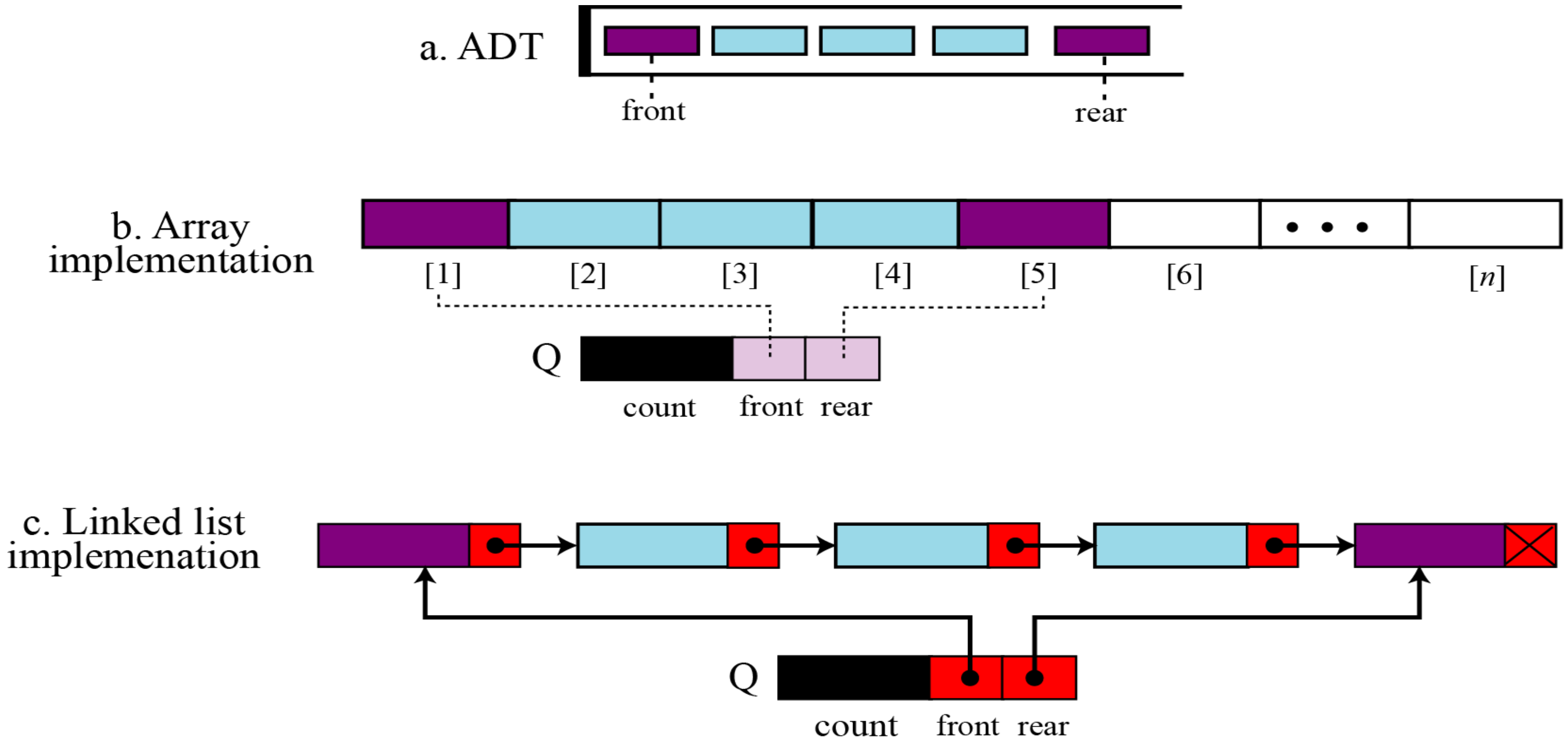
Queue ADT

Queue ADT

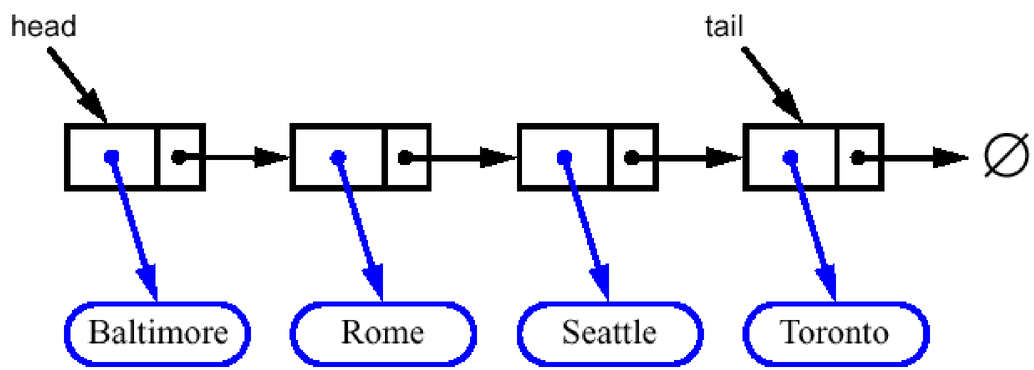
Definition	A list of data items in which an item can be deleted from one end, called the *front* of the queue and an item can be inserted at the other end, called the *rear* of the queue.
Operations	**queue:** Creates an empty queue.
	enqueue: Inserts an element at the rear.
	dequeue: Deletes an element from the front.
	empty: Checks the status of the queue.

Queue implementation

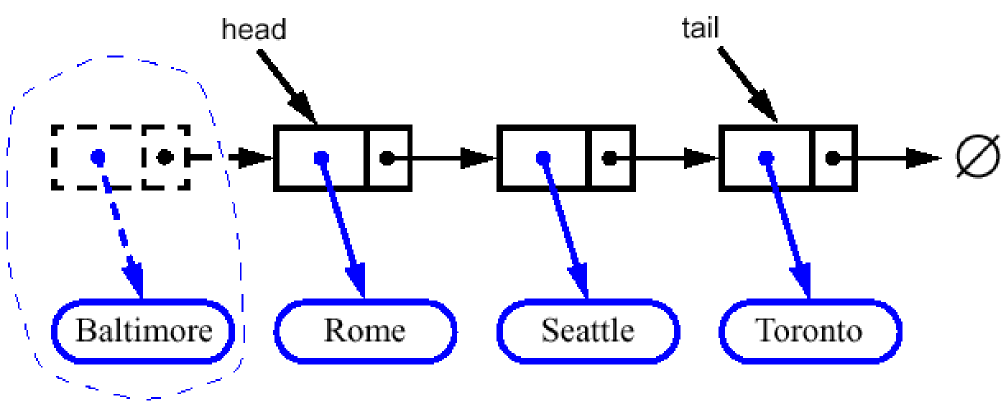
- A queue ADT can be implemented using either an array or a linked list



Queues: Linked List Implementation

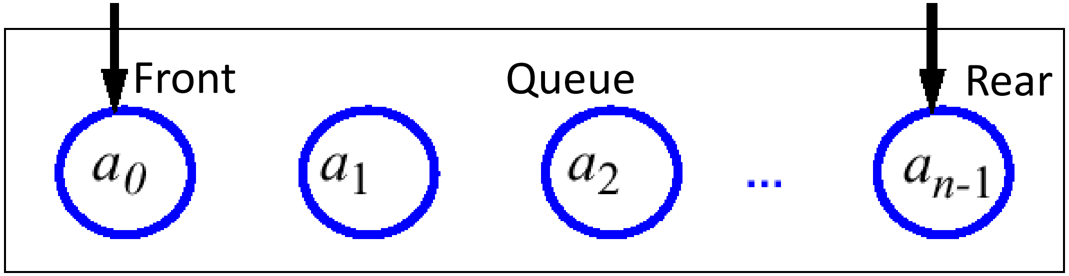


- Dequeue - advance head reference



Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)

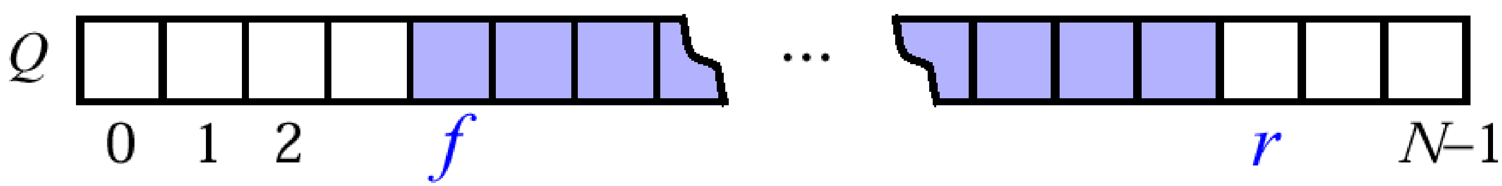


Queues

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)
  - Initially, f=r=0 and the queue is empty if f=r



Queues

Disadvantage

Repeatedly enqueue and dequeue a single element N times.

Finally, f=r=N.

- No more elements can be added to the queue, though there is space in the queue.

Solution

Let f and r wraparound the end of queue.

Queues: An Array Implementation

“wrapped around” configuration

- Each time r or f is incremented, compute this increment as (r+1)modN or (f+1)modN

Queues: An Array Implementation

- Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
    return Error
return Q[f]
```

```
Algorithm dequeue()
if isEmpty() then
    return Error
Q[f]=null
f=(f+1)modN
```

```
Algorithm enqueue(o)
if size = N - 1 then
    return Error
Q[r]=o
r=(r +1)modN
```

Establishing order of growth using the definition

Definition: $f(n)$ is in $O(g(n))$ if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n$ is $O(n^2)$

- $5n+20$ is $O(n)$

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(\text{n}))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq \text{b}$

- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then
$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{ order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{ order of growth of } g(n) \end{cases}$$

Examples:

- **$10n$ vs. n^2**
- **$n(n+1)/2$ vs. n^2**

L'Hôpital's rule and Stirling's formula

L'Hôpital's rule: If $lim_{n\to\infty} f(n) = lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

Example: log n vs. n

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Example: 2^n vs. $n!$