RECURSION

Recurrence Relation.pptx

Slide 18

What is Recursion?

- Recursion is a repetitive process in which an algorithm calls itself
- Each recursive call solves an identical, but smaller, problem.
- A test for the base case (initial condition) enables the recursive calls to stop

Recursive Algorithm : An Example

- Recall: Fibonacci Sequence

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

  $f_n = f_{n-1} + f_{n-2},\ n \geq 2$ $\qquad f_0=0,\ f_1=1$ (initial condition)

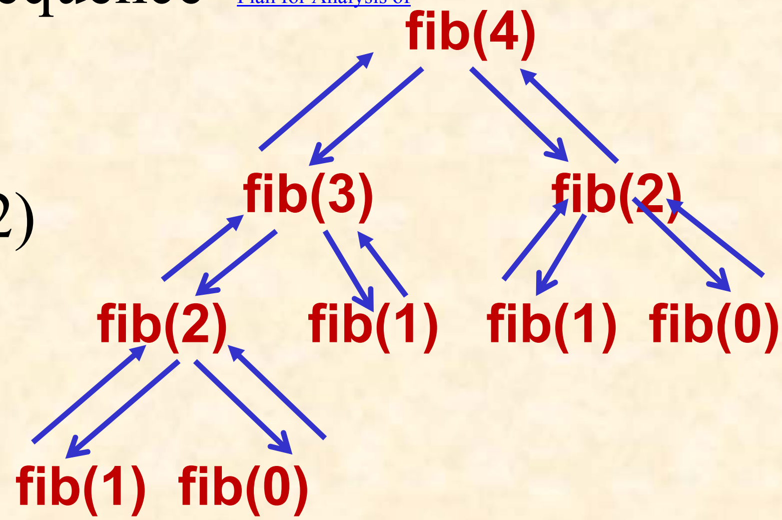
  $f_2 = f_1 + f_0 = 1+0 = 1$

  $f_3 = f_2 + f_1 = 1+1 = 2$

  $f_4 = f_3 + f_2 = 2+1 = 3$

Recursive algorithm for generating Fibonacci sequence Plan for Analysis of Recursive Algorithms

fibo_2(*n*)
  if (*n* == 0 or *n* == 1) return (*n*)
  else { *result* = *fibo_2*(*n*-1)+*fibo_2*(*n*-2)
      return (*result*)
    }



Recurrence Relations

- Fibonacci Sequence

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

  $f_n = f_{n-1} + f_{n-2},\ n \geq 2$ $\qquad f_0=0,\ f_1=1$ (initial condition)

  $f_2 = f_1 + f_0 = 1+0 = 1$

  $f_3 = f_2 + f_1 = 1+1 = 2$ $\qquad a_0, a_1, \ldots, a_{n-1}, a_n$

  $f_4 = f_3 + f_2 = 2+1 = 3$

- **Recurrence relation**: an equation that relates the n^{th} element, a_n, of a sequence to certain of its predecessors, $a_0, a_1, \ldots, a_{n-1}$.
  - We need an **initial condition** that provides the values for a finite number of elements of the sequence initially

Content of a Recursive Method

- **Base case(s)**
  - Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
  - Every possible chain of recursive calls **must** eventually reach a base case.
- ***Recursive calls***
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

How do I write a Recursive Function?

- Determine the input size
- Determine the base case(s)
  - the one for which the answer is known
- Determine the general case
  - the one where the problem is expressed as a smaller version of itself

- Example - the factorial function: Slide 5

  $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n\text{-}1) \cdot n \quad \text{for } n \geq 1$

  $0! = 1$ by definition ← **Base case**

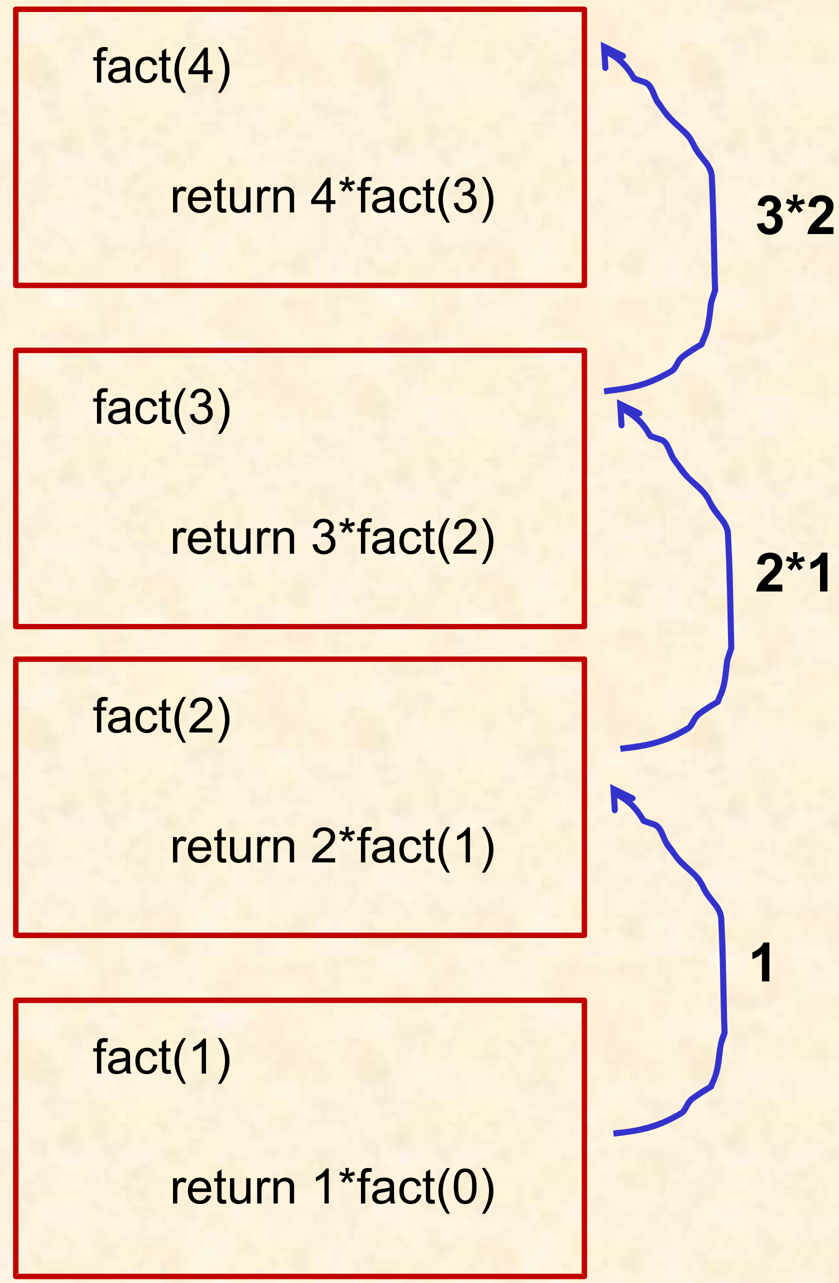
- Let $f(n) = n! = n(n\text{-}1)!$
- Hence, $f(n) = n \cdot f(n\text{-}1)$ ← **General case**
- Algorithm for factorial function: *fact(n)*

```
fact(n)
    if (n == 0) return 1;            //initial condition (base case)
    else return n * fact(n-1);       // recursive call
```

$fact(n)$

$\quad$ if (n == 0) return 1;

$\quad$ else return n * $fact(n$-1);



T(n) = number of multiplications needed to compute factorial n

T(0) = 0

T(1) = 1

T(2) = T(1) + 1 = 1 + 1 = 2

T(3) = T(2) + 1 = 2 + 1 = 3

T(4) = T(3) + 1 = 3 + 1 = 4

- Time analysis for the recursive algorithm of the factorial function
  - Let $T(n)$ be the number of multiplications needed to compute $fact(n)$
  - $T(n) = T(n\text{-}1) + 1$
    - $T(n\text{-}1)$ multiplications are needed to compute $fact(n\text{-}1)$, and one more multiplication is needed to multiply the result by n
    - Note: $T(0) = 0$
- Using the method of **backward substitutions**,

$$
\begin{aligned}
T(n) &= T(n\text{-}1) + 1 \\
&= [T(n\text{-}2) + 1] + 1 = T(n\text{-}2) + 2 \\
&= [T(n\text{-}3) + 1] + 2 = T(n\text{-}3) + 3 \\
&\dots \\
&= [T(n\text{-}n) + 1] + (n\text{-}1) = n
\end{aligned}
$$

T(n-1) = T(n-2) + 1
T(n-2) = T(n-3) + 1

- Time efficiency of the recursive algorithm is of $O(n)$

Plan for Analysis of Recursive Algorithms

1. Decide on parameter *n* indicating *input size*
2. Identify algorithm's *basic operation*
3. Determine *worst* case for input of size *n*
   – May also need to determine the *average* and *best* cases
4. Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic operation is executed
5. Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method. Slide 4

Example: Summing elements in an Array

Algorithm LinearSum(A, n):

Input:

A integer array A and a positve integer n, such that A has at least n elements

Output:

The sum of the first n integers in A

if $n = 1$ **then**
  return $A[1]$
else
  return LinearSum(A, n - 1) + $A[n]$

	1	2	3	4
A	5	3	2	6

Example recursion trace:

return 10 + A[4] = 10 + 6 =16

LinearSum(A, 4)

return LinearSum(A, 3) + A[4]

LinearSum(A, 3)

return 8 + A[3] = 8 + 2 =10

return LinearSum(A, 2) + A[3]

LinearSum(A, 2)

return LinearSum(A, 1) + A[2]

return 5 + A[2] = 5 + 3 =8

LinearSum(A, 1)

return A[1]

return A[1] = 5

Time analysis for the recursive algorithm to compute the sum of the elements in an array

Let $T(n)$ be the number of additions needed to compute LinearSum(A, n)

$T(n) = T(n\text{-}1) + 1$

$T(n\text{-}1)$ additions are needed to compute LinearSum(A, n -1), and one more addition is needed to add the result with A[n]

Note: $T(0) = 0$

=> same time complexity as the factorial algorithm

Example: Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray($A, i + 1, j - 1$)

return

ReverseArray(A,1,4)

	1	2	3	4
A	**6**	3	2	**5**

↓

ReverseArray(A,2,3)

	1	2	3	4
A	6	**2**	**3**	5

↓

ReverseArray(A,3,2)

	1	2	3	4
A	5	3	2	6

Time analysis for the Reverse Array Algorithm

– Let $T(n)$ be the number of swapping operations needed to reverse the elements of an array of size n

– $T(n) = T(n\text{-}2) + 1$ **why ??**

To reverse an array with n elements : we need to reverse a subarray with n-2 elements and perform one swapping operation

$= [\ \mathbf{T(n-4) + 1}\ ] + 1$

$= T(n-4) + 2$

$= [\ \mathbf{T(n-6) + 1}\ ] + 2$

$= T(n-6) + 3$

$= T(n-n) + n/2$

$= T(0) + n/2$

Time efficiency of the recursive algorithm is of $O(n)$

Example: Computing Powers

- $x^n = x \cdot x^{n-1}$
- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \textit{otherwise} \end{cases}$$

- Recursive algorithm for power function: $p(x,n)$

```
if (n == 0) return 1;          //initial condition
else return x * p(x,n-1);      // recursive call
```

- Similar to the factorial function, this leads to an power function that runs in $O(n)$ time
- We can do better than this, however.

Recursive Squaring [Recursive Squaring.pptx]

- Note: if n is even, $x^n = (x^{n/2})^2$ $\quad (x^{n/2})^2 = (x^{n/2})(x^{n/2}) = x^n$

  if n is odd, $x^n = x \cdot (x^{(n-1)/2})^2$ $\quad (x^{(n-1)/2})^2 = (x^{(n-1)/2})(x^{(n-1)/2}) = x^{n-1}$

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot (p(x,(n-1)/2))^2 & \text{if } n > 0 \text{ is odd} \\ (p(x,n/2))^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,
  - $p(10,4) = (p(10,2))^2 = (\,(p(10,1))^2\,)^2$
    $= ((\,10 * (p(10,0))^2\,)^2\,)^2$
    $= ((\,10 * 1^2)^2)^2$
    $= ((10)^2)^2 = (100)^2 = 10000$

A Recursive Squaring Method

- **Algorithm** Power(x, n):
- **Input:** A number x and integer $n \geq 0$
- **Output:** The value x^n

  if n = 0 **then**

    return 1

  if n is odd **then**

    y = Power(x, (n -1) / 2)

    return $x \cdot y \cdot y$

  else

    y = Power(x, n/ 2)

    return $y \cdot y$

Note: it is important that we use the variable *y* twice instead of calling the function Power(x,n) twice.

A Recursive Squaring Method

- **Algorithm** Power(x, n):
- **Input:** A number x and integer $n \geq 0$
- **Output:** The value x^n

```
    if n = 0 then
        return 1
    if n is odd then
        y = Power( x, ( n -1) / 2)
        return x · y ·y
    else
        y = Power( x, n/ 2)
        return y · y
```

T(n) = number of multiplications to compute x^n

$T(n) \leq T(n/2) + 2$

Power(2,6)

else
y = power(2,3)
return $2^3 * 2^3$

Power(2,3)
if n is odd **then**
y = power(2,1)
return 2*2*2

Power(2,1)
if n is odd **then**
y = power(2,0)
return 2*1*1

T(6)

$\leq T(3) + 2$

$= (T(1) + 2) + 2$

$= (T(0) + 2) +2+ 2$

Note:

T(1) = T(0) + 2

= 2

Time Analysis of the Recursive Squaring Method

- Each time the recursive call is made
  - The value of n is halved
  - There are at most two multiplications
- $T(n) \le T(n/2) + 2$

  $\le T(n/2^2) + 2\times 2$

  $\le T(n/2^3) + 3\times 2$

  …

  $\le T(n/2^i) + i\times 2$

$T(n/2) = T(n/2^2) + 2$

$T(n/2^2) = T(n/2^3) + 2$

$T(n) \le T(1) + 2\lg n$

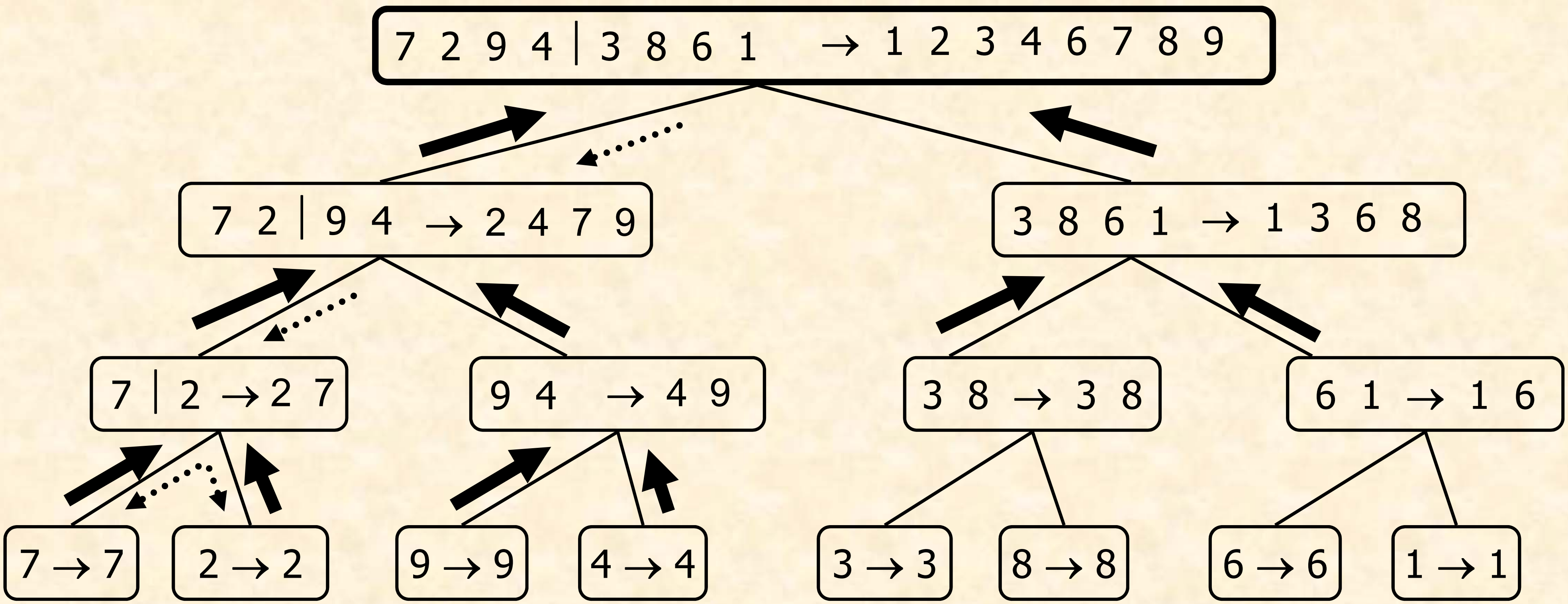
$= 2 + 2 \lg n$

- The base case is reached when $2^i = n \Rightarrow i = \lg n$
- Hence, $T(n) \le 2 + 2 \lg n$
  - The running time is of $O(\lg n)$ which is a big improvement over the previous algorithm with running time $O(n)$

Example: Merge Sort

- Merge-sort on an input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$ elements consists of three steps:
  - Partition $\boldsymbol{S}$ into two sequences $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ of about $\boldsymbol{n}/2$ elements each
  - Recursively sort $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Merge $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a unique sorted sequence

Execution Example (cont.)



Example: Merge Sort (cont.) mergesort.pptx

Algorithm ***mergeSort*(*S*, *i*, *j*)**

Input sequence ***S*** with ***n*** elements, and indices *i* and *j*

Output sorted sequence ***S***

if ($i == j$) **return**

else

$m = (i+j)/2$

***mergeSort*(*S*, *i*, *m*)**

***mergeSort*(*S*, *m*+1, *j*)**

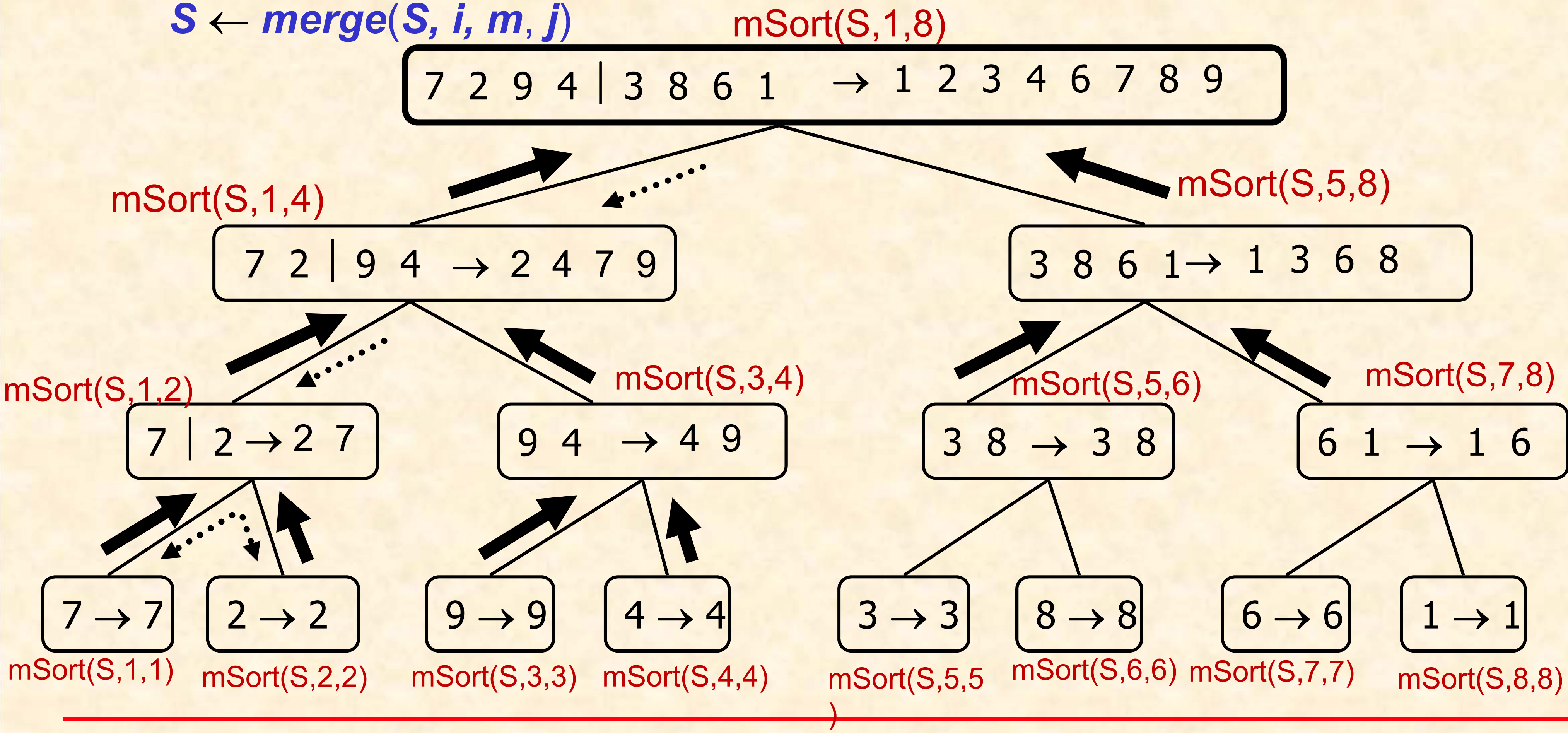
$S \leftarrow$ ***merge*(*S*, *i*, *m*, *j*)**

Let T(n) = time to sort n elements

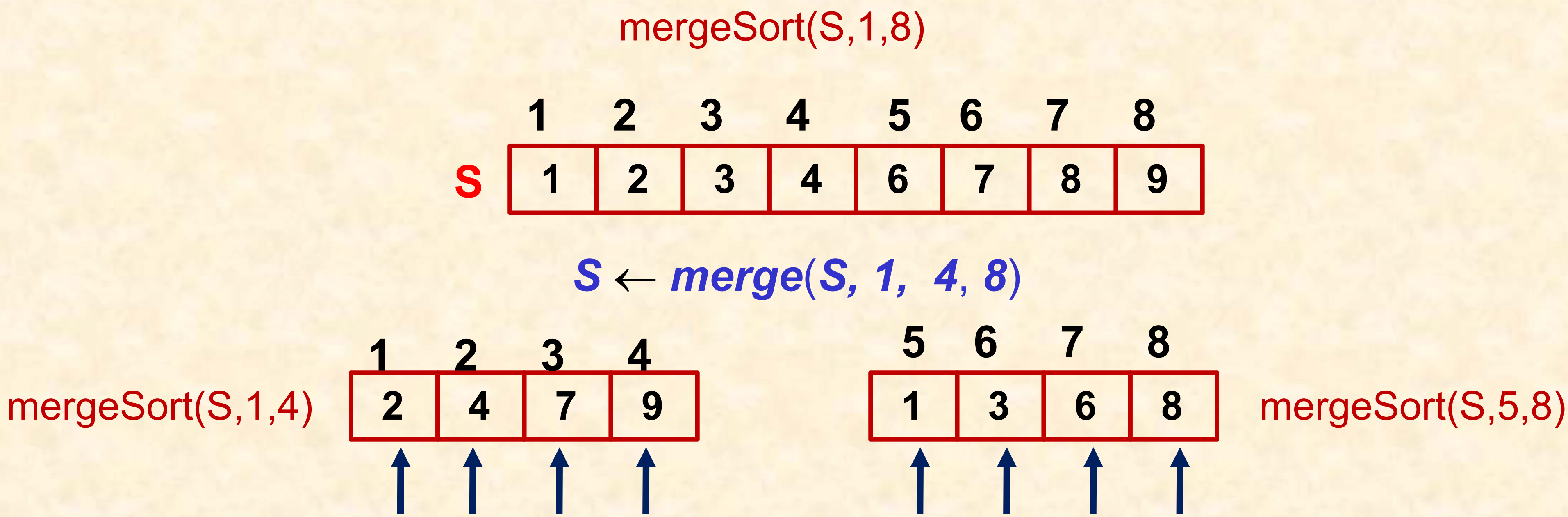
$T(n) = T(n/2) + T(n/2) + bn$

***mergeSort*(*S*, *i*, *j*)**

if (*i* == *j*) **return**
else
  ***m* = (*i*+*j*)/2**
  ***mergeSort*(*S*, *i*, *m*)**
  ***mergeSort*(*S*, *m*+1, *j*)**
  S ← ***merge*(*S*, *i*, *m*, *j*)**



```
if (i == j) return
else
    m = (i+j)/2
    mergeSort(S, i, m)
    mergeSort(S, m+1, j)
    S ← merge(S, i, m, j)
```



Example: Merge Sort (cont.)

mergesort.pptx

Algorithm ***mergeSort*(*S*, *i*, *j*)**

if $(i == j)$ **return**

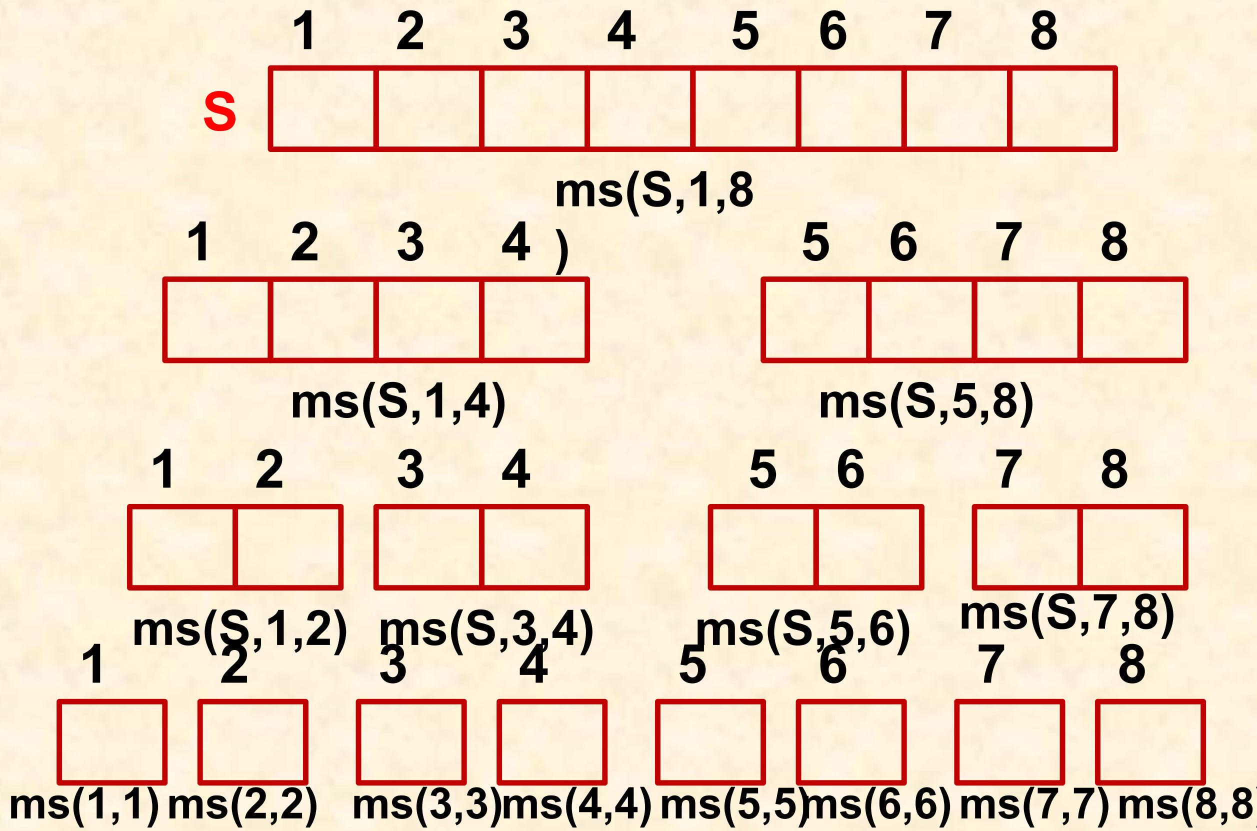
else

$m = (i+j)/2$

mergeSort(*S*, *i*, *m*)

mergeSort(*S*, *m*+1, *j*)

$S \leftarrow$ *merge*(*S*, *i*, *m*, *j*)



T(8)

T(4) + T(4) + 8b

T(2) + T(2) + 4b　　T(2) + T(2) + 4b

T(1) + T(1) + 2b　　T(1) + T(1) + 2b　　T(1) + T(1) + 2b　　T(1) + T(1) + 2b

$T(2) = 2\,T(1) + 2b$

$T(4) = 2T(2)+4b = 2[2\,T(1) + 2b] + 4b$

$= 2^2\,T(1) + 2^3 b$

$T(8) = 2T(4) + 8b = 2\,[2^2\,T(1) + 2^3 b] + 8b$

$= 2^3\,T(1) + 24b$

Time Analysis: Merge Sort

- The last step of merging two sorted sequences, each with $\boldsymbol{n}/2$ elements, takes at most $\boldsymbol{bn}$ steps, for some constant $\boldsymbol{b}$.
- Likewise, the base case ($\boldsymbol{n} < 2$) will take at most $\boldsymbol{b}$ steps.
- Therefore, if we let $\boldsymbol{T(n)}$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

Time Analysis: Merge Sort (cont.)

$$T(n) = 2T(n/2) + bn$$

$$T(n/2) = 2T(n/2^2) + b(n/2)$$

$$= 2(2T(n/2^2) + b(n/2)) + bn$$

$$T(n/2^2) = 2T(n/2^3) + b(n/2^2)$$

$$= 2^2 T(n/2^2) + 2bn$$

$$= 2^2(2T(n/2^3) + b(n/2^2)) + 2bn$$

$$= 2^3 T(n/2^3) + 3bn$$

$$= \ldots$$

$$= 2^i T(n/2^i) + ibn$$

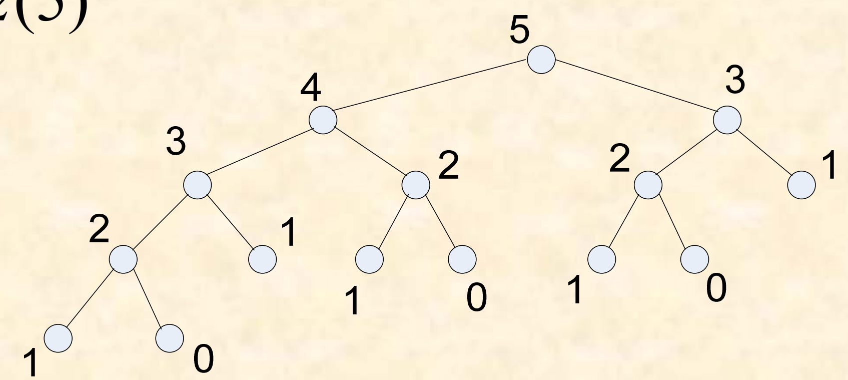
- Note that base case, $T(1)=b$, occurs when $n=2^i$. That is, $i = \lg n$.
- So, $T(n) = 2^{\lg n} b + bn \lg n = bn + bn \lg n$
- Thus, $T(n)$ is $O(n \lg n)$.

❖ One should be careful with recursive algorithms as their conciseness, clarity and simplicity may hide their inefficiency

- Recall that the Fibonacci numbers are defined recursively: $f_n = f_{n-1} + f_{n-2}$.
  - We have shown an iterative algorithm (algorithm 1: fibo_1) for the Fibonacci numbers with running time of $O(n)$
- The recursive algorithm: *fibo_2(n)*

```
fibo_2(n)
    if (n == 0  or  n == 1)  return (n)
    else { result = fibo_2(n-1)+fibo_2(n-2)
           return (result)
         }
```

- By using recursive call, each call leads to 2 further calls, e.g. for *fibo_2*(5)



- The total number of calls grows exponentially
- Computing the running time of recursive *fibo_2*
  - We have :

$$T(n) = \begin{cases} O(1) & n < 2 \\ T(n-1) + T(n-2) + O(1) & n \geq 2 \end{cases}$$

  - It can be shown that

$$T(n) = \Omega((\frac{3}{2})^n)$$

- The recursive algorithm is clearly impractical
  - the non-recursive algo. has only linear running time *O(n)* [Recursion – Fibonacci Sequence](#)

Basic Data Structures - Stacks and Queue

Lec11_Begin.pptx Singly Linked Lists

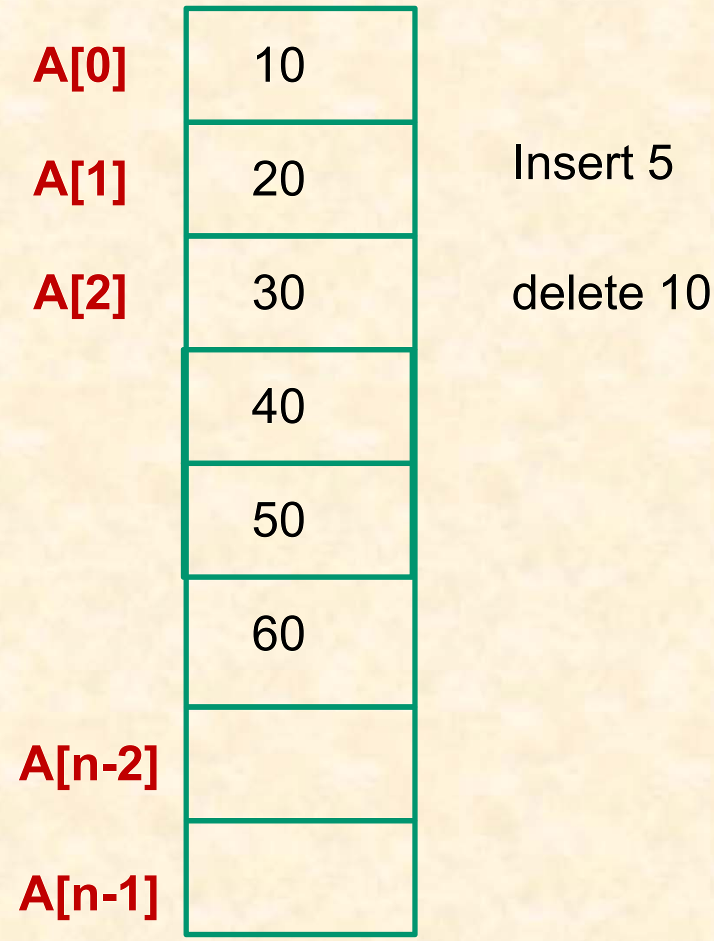
Data Structures [Basic Data Structures]

- Recall: A *data structure* is data together with structural relations on data to promote efficient processing of the data.

- Examples [Array.pptx]
  - Array: provides random access to data at constant access time
  - Linked list: supports easy insertion and deletion of data

Array

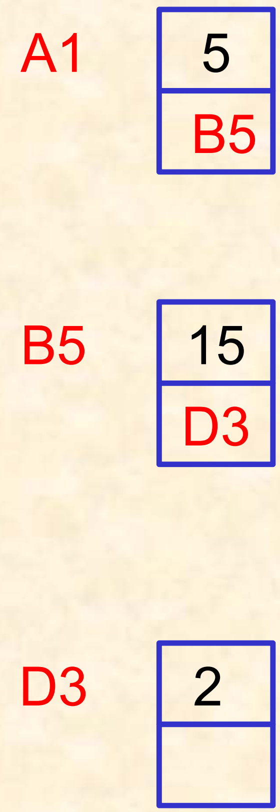
- A group of data elements stored in contiguous memory.
- Advantage: Instant access to any element in the array (ie it is just as easy to retrieve the value of the first element as any other).
- Disadvantages:
  - Fixed size
  - Hard to insert new elements in the middle of the array
  - Deletion may be time consuming.



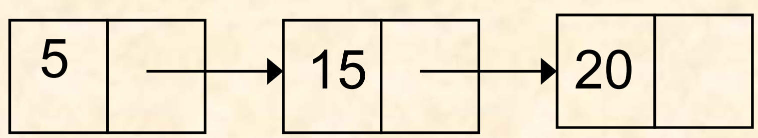
Basic Data Structures

[What is the use of Data Structure ???]

- Linked List
  - elements are stored in memory locations that are dynamically allocated
  - each element of a linked list is composed of two parts:
    - **the data value**
    - **the address of the next element in the list**



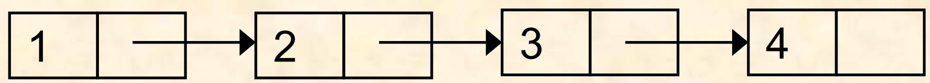
Input: 5, 15, 20



Linked Lists

- **Each element of a linked list is composed of two parts:**
  - **The data value (can be any type)**
  - **A pointer to the next element in the list (type must be a pointer)**

- **Elements (n**odes) are not stored consecutively in memory, but instead are allocated *dynamically*



Advantages:

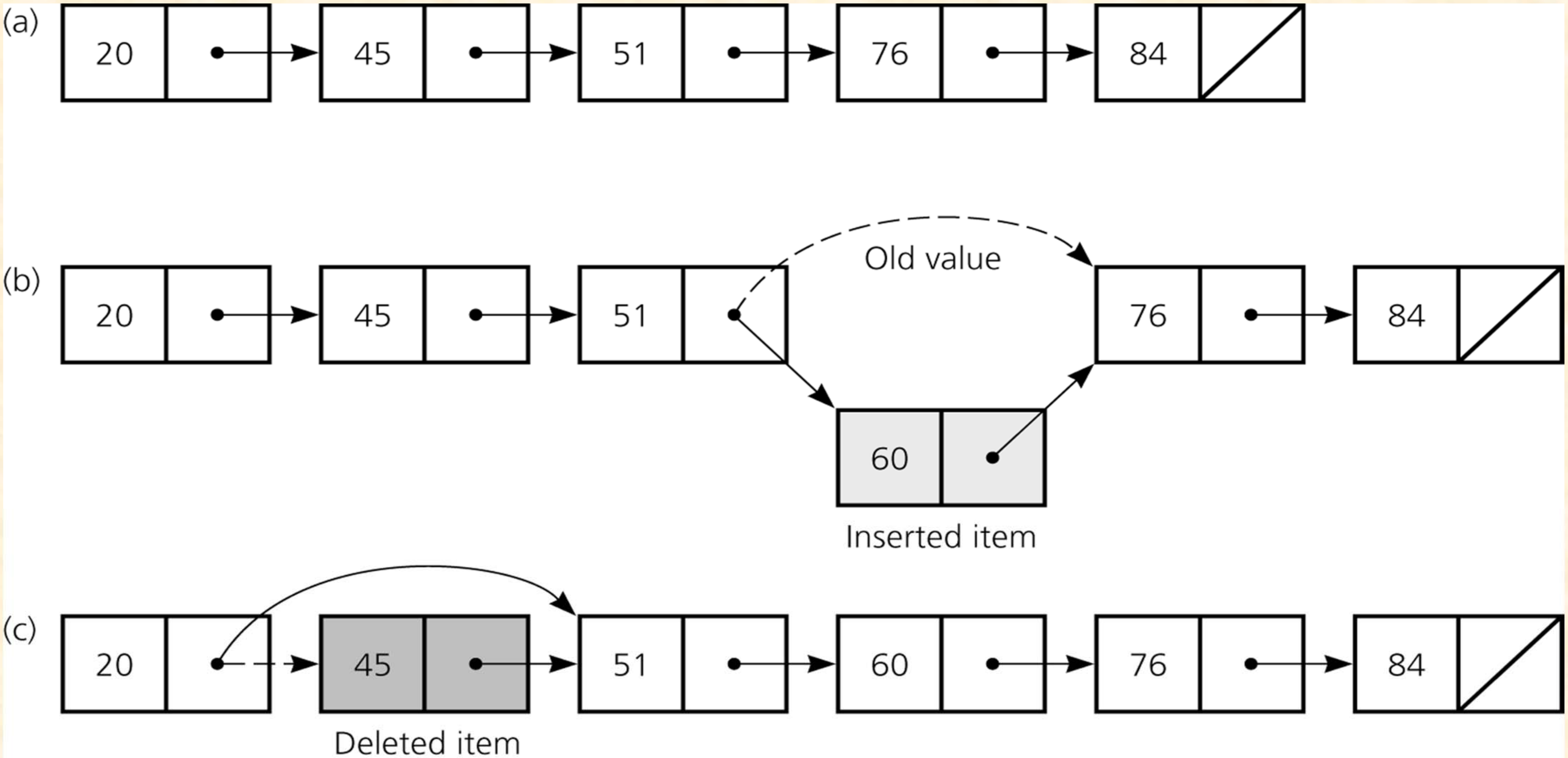
The linked list can change size easily.

Elements can be inserted and deleted easily into linked lists.

Disadvantage:

You do not have quick access to members

a) Linked List (b) Insertion (c) Deletion



Array and Linked List

- Array
  - ✓ Random access of array element at constant time
  - ✗ Fixed size
  - ✗ Insertion and deletion of elements take time $\Theta(n)$
    - require moving of possibly n items for the insertion of an item and after deletion of an item
- Linked List
  - ✓ Storage location can be allocated as and when needed
  - ✓ Simple insertion and deletion operations at constant time
  - ✗ Access time varies depending on the number of items in the list, the structure of the list and the location of an item
    - require time $\Theta(n)$ in the worst case

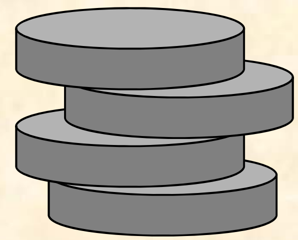
Abstract Data Types

- Some data structures are described ***abstractly*** in terms of the operations on the data but their implementations are not specified

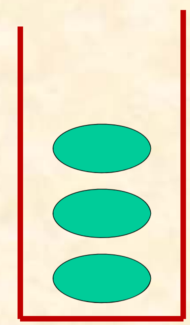
- *Abstract data type* (*ADT*) consists of data together with functions that operate on the data.
  - Only the behavior of the functions is specified.
  - How the functions are implemented are not specified.
  - E.g. stacks and queues

Stack

- A stack is a data structure in which an item can only be inserted and removed from one end (the top).
  - It follows LIFO strategy



- Applications of stacks
  - Page-visited history in a Web browser
    - Each time a site is visited, its address is "***pushed***" onto the stack.
    - A user can "***pop***" back to the previously visited sites using the *back* button.
  - Undo sequence in a text editor
    - Editing commands are stored in a stack with the most recently used command on the top.



Stack ADT

- Main functions:
  - stack_init( ): Make the stack empty.
  - empty( ): Return *true* if the stack is empty; otherwise, return *false*.
  - push(*val*): Add the item *val* to the top of the stack.
  - pop( ): Remove the top (most recently added) item from the stack and no value is returned.
  - top( ): Return the item most recently added to the stack, but do not remove it.

- Note that the ADT defines the functions of a stack but does not specify how to implement the stack.

- Example: The following shows a series of operations and their effects on the stack *S*.

Operation	Output	Bottom – Stack – Top
S.stack_init( )	-	( )
S.push(5)	-	(5)
S.push(3)	-	(5, 3)
S.pop( )	-	(5)
S.push(7)	-	(5,7)
S.pop()	-	(5)
S.top( )	5	(5)
S.pop( )	-	( )
S.pop( )	"error"	( )
S.empty( )	*true*	( )
S.push(9)	-	(9)

Implementing Stack using Array

- A simple way of implementing the Stack uses an array
- We add elements from left to right
- A variable *t* keeps track of the index of the top element (size of stack is *t*+1)
  - When an item is pushed onto the stack, we increment *t* and put the item in the cell at index *t*.
  - When an item is popped off the stack, we decrement *t*.

Algorithms for Stack functions (using Array)

```
stack_init() {
    t = -1
}
```

```
empty() {
    return t == -1
}
```

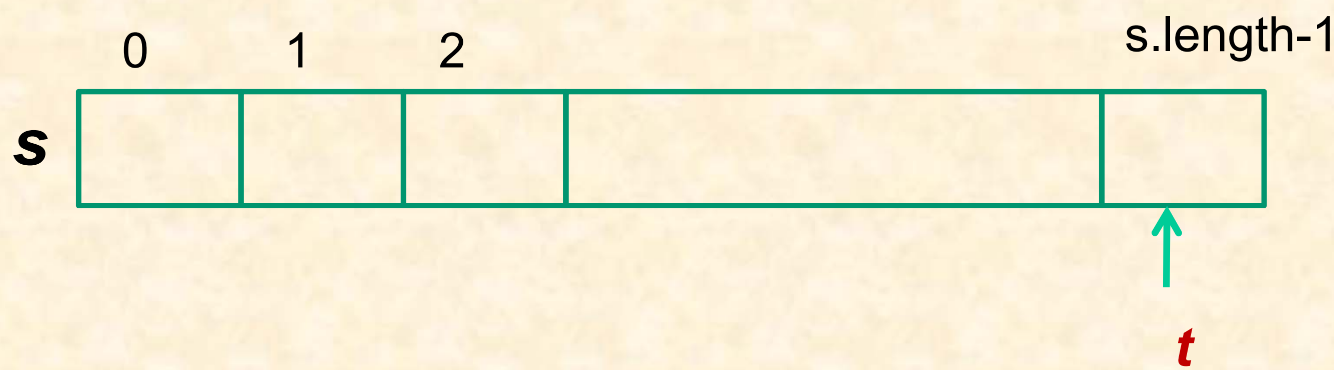
```
pop() {
    if S.empty( ) then
        throw EmptyStackException
    else
        t = t – 1
}
```

```
top() {
    if S.empty( ) then
        throw EmptyStackException
    else
        return S[t]
}
```

Implementation of Stack: Error Checking

- The array storing the stack elements may become full
- A push operation will then throw a *FullStackException*
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
push(val) {
    if  t==S.length-1 then
        throw FullStackException
    else
        t = t + 1
        S[t] = val
}
```



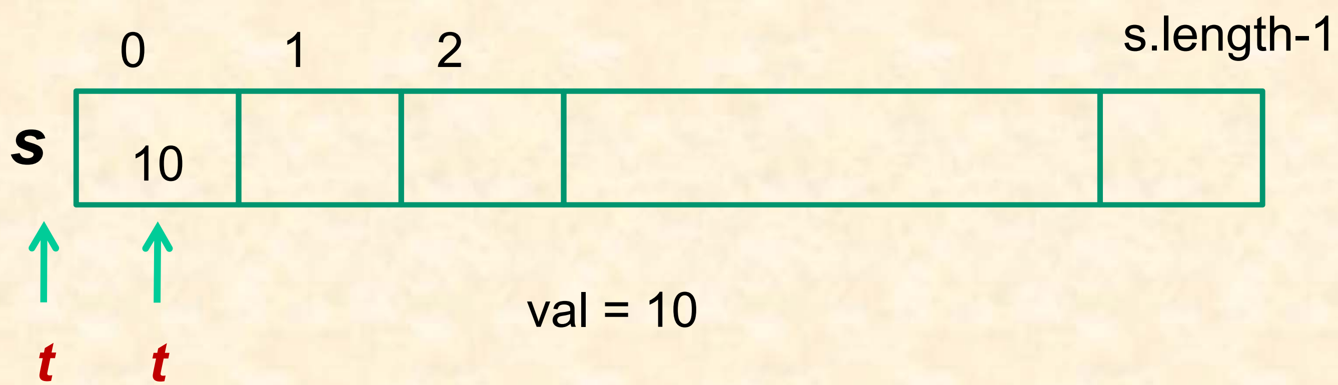
Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Example: This algorithm returns *true* if there is exactly one element on the stack. The code uses the abstract data type stack.

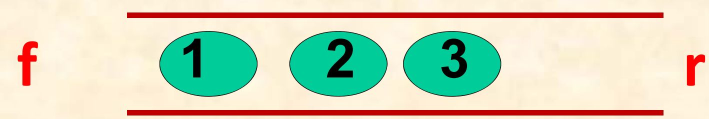
Input Parameter: *s* (the stack)
Output Parameter: None

```
one(s) {
    if (s.empty())
        return false
    val = s.top()
    s.pop()
    flag = s.empty()
    s.push(val)
    return flag
}
```

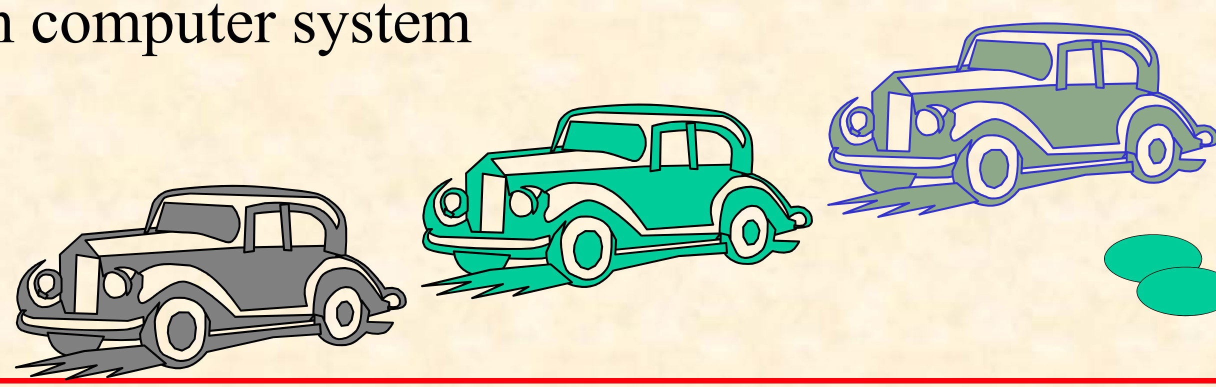


Queue

- Insertions are at the rear of the queue and removals are at the front of the queue
- Insertion (enqueue) and deletion (dequeue) follow the first-in first-out (FIFO) scheme



- Applications:
  - Waiting lines: handling of transaction requests
  - Access to shared resources (e.g. printer)
  - Multi-processing in computer system



Queue ADT

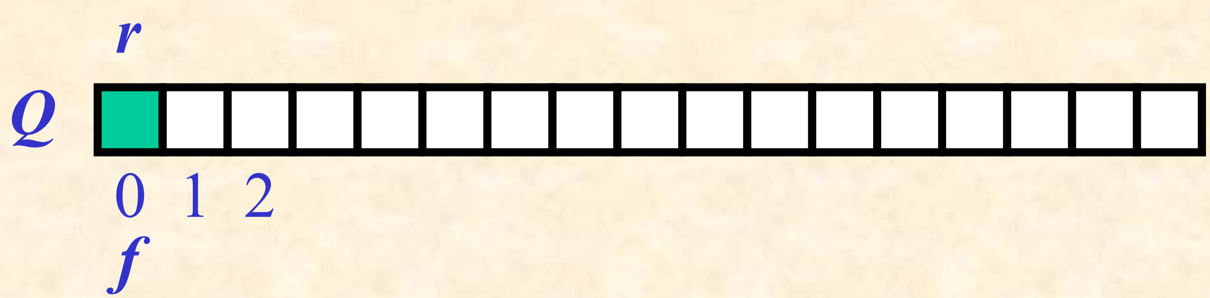
- Main functions:
  - queue_init( ): Make the queue empty.
  - empty( ): Return *true* if the queue is empty; otherwise, return *false*.
  - enqueue(*val*): Add the item *val* to the rear of the queue.
  - dequeue( ): Remove the item from the front (least recently added) of the queue. No value is returned.
  - front( ): Return the item from the front of the queue, but do not remove it.

Example: The following shows a series of operations and their effects on the queue *Q*.

Operation	Output	front ← *Q* ← rear
Q.queue_init( )	-	( )
Q.enqueue(5)	-	(5)
Q.enqueue(3)	-	(5, 3)
Q.dequeue( )	-	(3)
Q.enqueue(7)	-	(3, 7)
Q.dequeue( )	-	(7)
Q.front( )	7	(7)
Q.dequeue( )	-	( )
Q.dequeue( )	"error"	( )
Q.empty( )	*true*	( )
Q.enqueue(9)	-	(9)

Queue

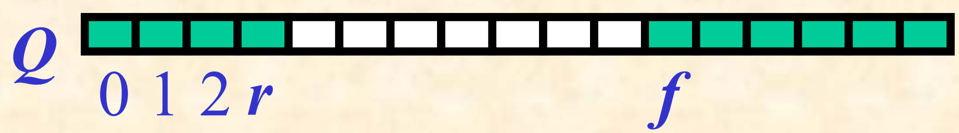
- Like a stack, a queue can be implemented using an array.
- Items are added at the rear and deleted from the front
  - need two variables, *r* and *f*, to track the indexes of the rear and front of the queue.
  - An empty queue has both *r* and *f* equal to -1.
- When an item is added to an empty queue, we set *r* and *f* to 0 and put the item in the cell 0.



- When an item is added to a nonempty queue Queue
  - we increment *r* and put the item in the cell *r*
  - an array of size ***N*** is used in a circular fashion
    - If *r* is the index of the last cell in the array, we "wrap around" by setting *r* to 0
    - Done by using the modulo operation, *r* mod *N*
  - If *r* = *f (after incrementing r)*, the queue is full and no item can be added **Why?**
- After an item is removed from a nonempty queue, we increment *f*.
  - Similarly, if *f* is the index of the last cell in the array, we "wrap around" by setting *f* to 0.
  - If *f* = *r (before incrementing f)*, the queue is empty (after deletion) ; Set *f* = *r* = -1 **Why?**

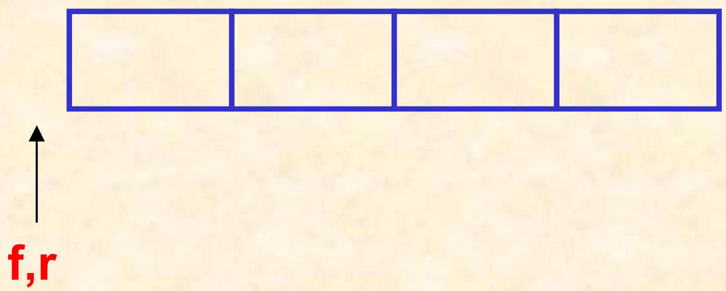
normal configuration

wrapped-around configuration

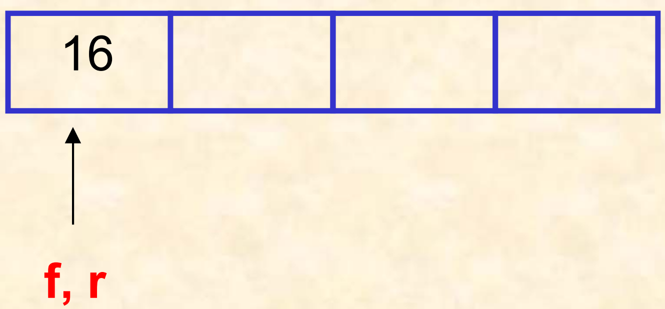


Queue

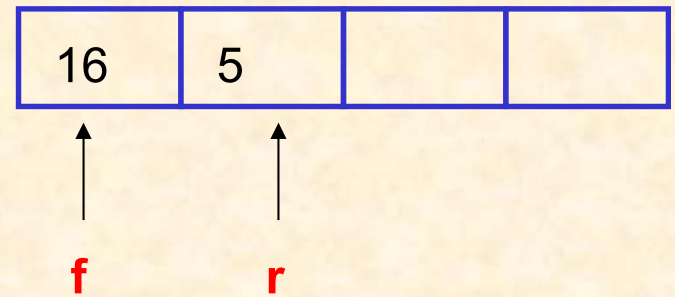
q.queue_init()



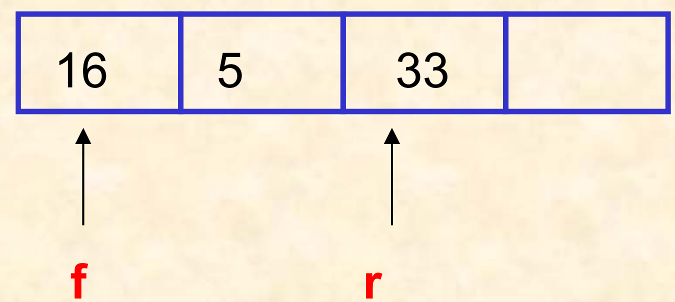
q.enqueue(16)



q.enqueue(5)

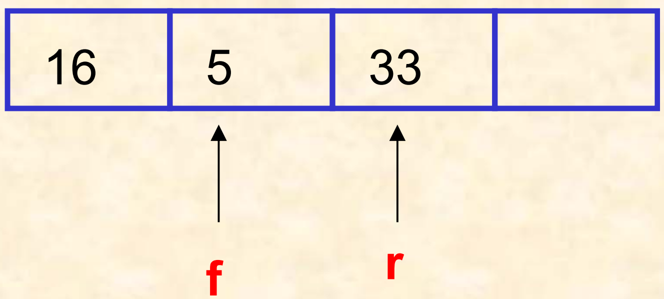


q.enqueue(33)

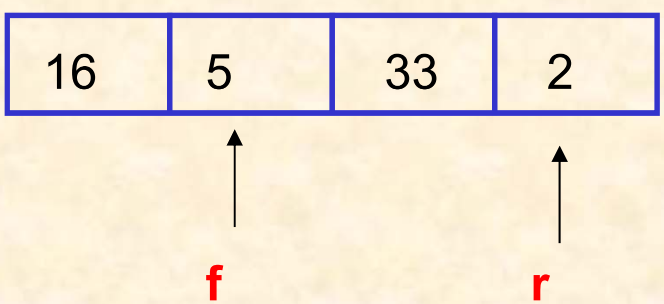


Queue

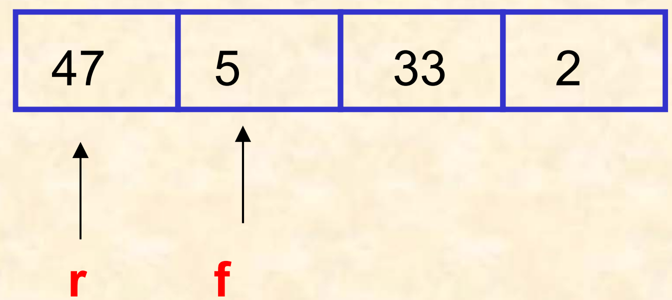
q.dequeue()



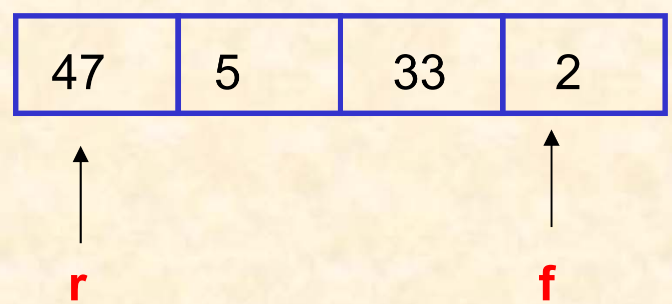
q.enqueue(2)



q.enqueue(47)



q.dequeue q.dequeue()



Queue

Slide 41

q.enequeue(12)

47	12	33	2

r

f

q.dequeue

47	12	33	2

r

f

q.dequeue q.dequeue()

47	12	33	2

r

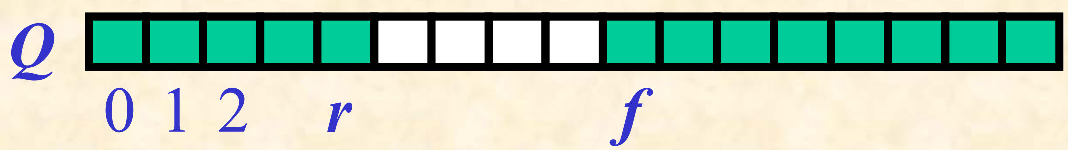
f

Algorithms for Queue functions

```
queue_init() {
    r = f = - 1
}
```

```
empty() {
    return r == - 1
}
```

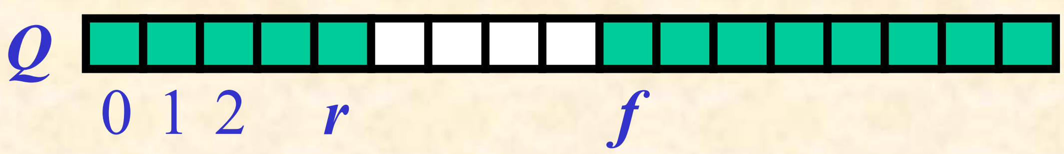
```
enqueue(val) {
    if (empty())
        r = f = 0
    else {
        r = r + 1
        if (r == Q.size) r = 0
        if r == f
            throw FullQException
    }
    Q[r] = val
}
```



Algorithms for Queue functions (cont.)

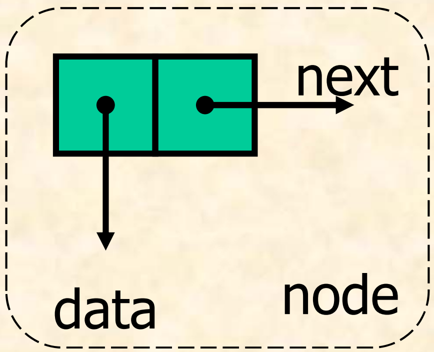
```
dequeue() {
    if (empty())
        throw QEmptyException
    else {
        //does queue contain one item?
        if (r == f)
            r = f = -1
        else {
            f = f + 1
            if (f == Q.size) f = 0
        }
    }
}
```

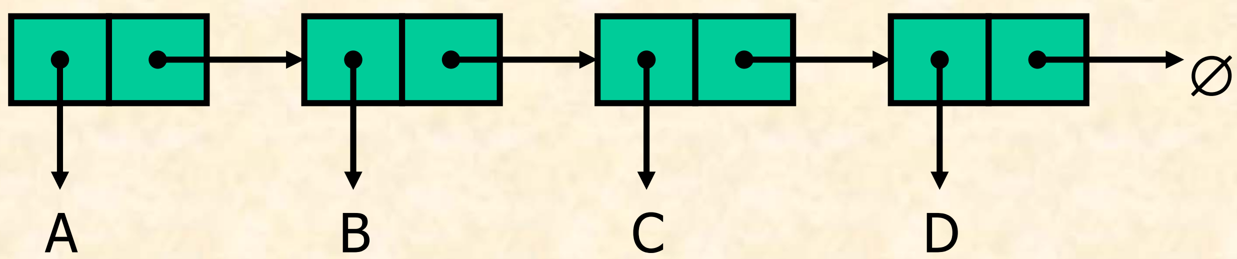
```
front() {
    if (empty())
        throw QEmptyException
    else return  Q[f]
}
```



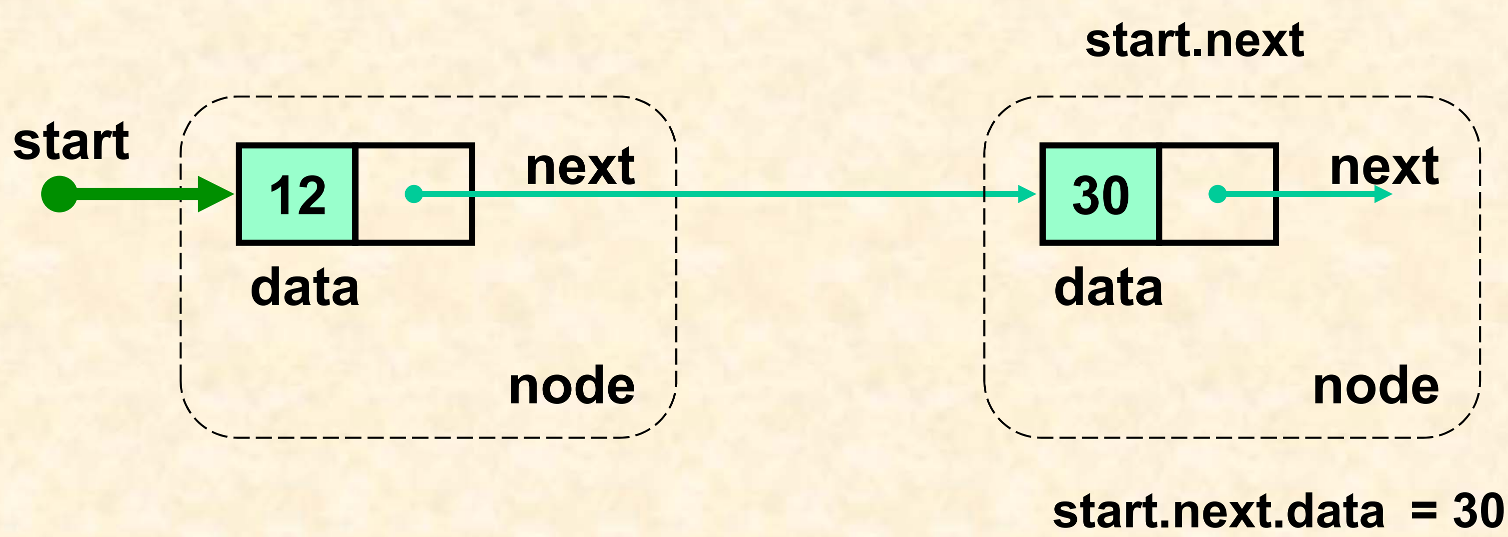
Singly Linked Lists

- Singly Linked list: a data structure consisting of a sequence of nodes
- a ***node*** consists of a data field *data* and a field *next* that links to the next node in the list.
  - When this structure is implemented in a computer, the next field is typically the address of the next node.
  - The next field of the last node has a special value ***null***



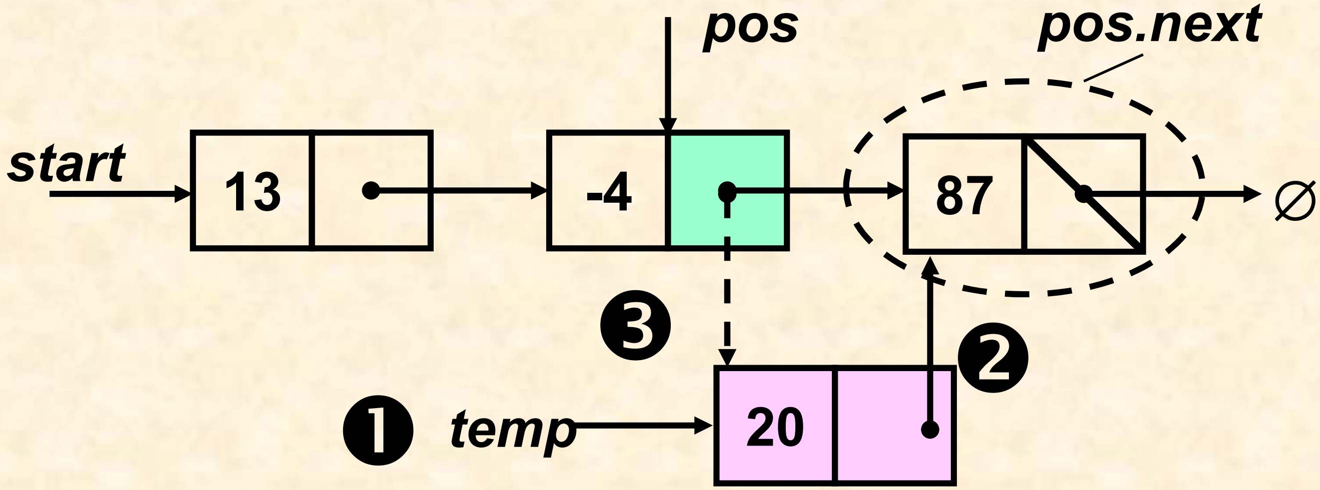


- Making reference to a field in a node by using the ***Dot*** notation Example: Printing the Data in a Linked List
- Example
  - *start* references to the first node in the example below
  - *start.data* references the data field (12)
  - *start.next* references the next field which contains "address" of the next node



Node Insertion

- Original list: 13, -4, 87
- Insert a new node with value 20 after -4
- *start* points to the beginning of the list, *pos* references the node that precede the new node, and *temp* references the new node to be inserted



1. A new node, *temp*, is obtained. Then data 20 is inserted at *temp.data*.
2. *temp.next* is set to *pos.next*.
3. Finally, *pos.next* is set to *temp*.

Node Insertion Algorithm

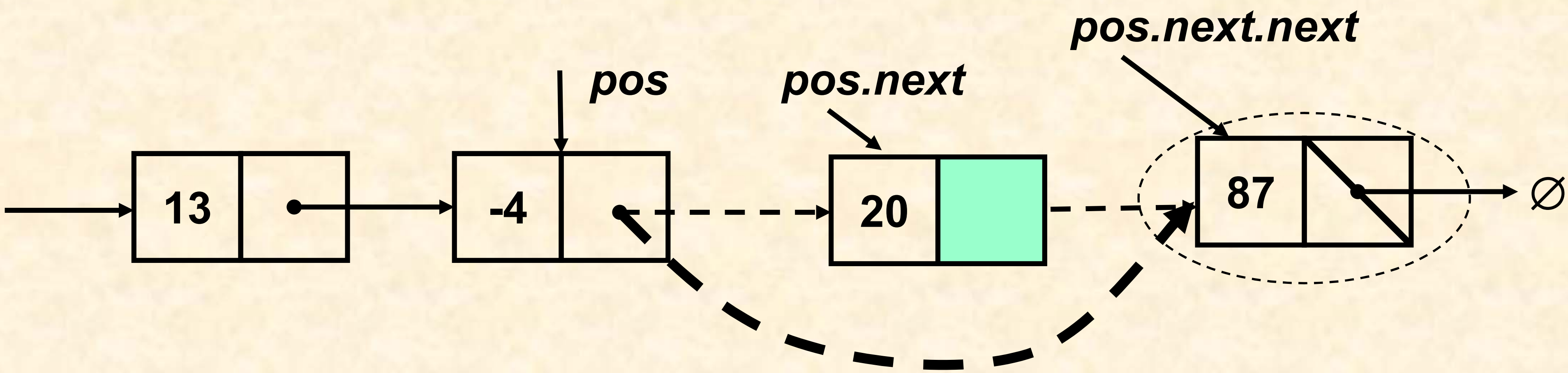
- This algorithm inserts the value *val* after the node referenced by *pos*.
  - The operator, *new*, is used to obtain a new node.

```
insert(val,pos) {
    temp = new node          } 1
    temp.data = val
    temp.next = pos.next       2
    pos.next = temp            3
}
```

- Note: the *insert* operation runs in constant time

Node Deletion

- Original list: 13, -4, 20, 87
- *pos* references the node preceding the node to be deleted
- Node is deleted by setting *pos.next* to *pos.next.next*



Assume that the list is not empty

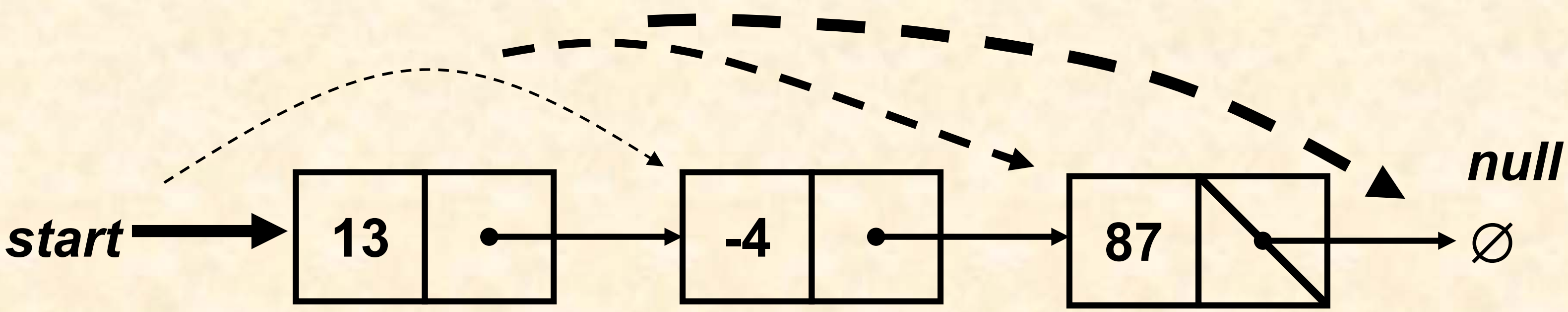
```
delete(pos) {
    pos.next = pos.next.next
}
```

The algo. *delete* also runs in constant time.

Example: Printing the Data in a Linked List

- The first node is referenced by *start*.

```
print(start) {
    while (start != null) {
        print-data (start.data);
        start = start.next
    }
}
```



- The alog. runs in time $\theta(n)$ when the linked list contains n nodes.

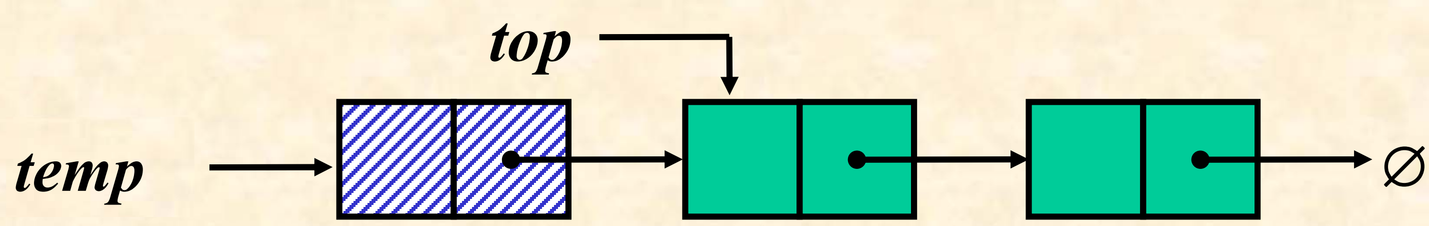
Example: Implementing Stack using Linked List

- The first node is referenced by *top*, the top of a stack.
- An empty stack has *top* = *null*

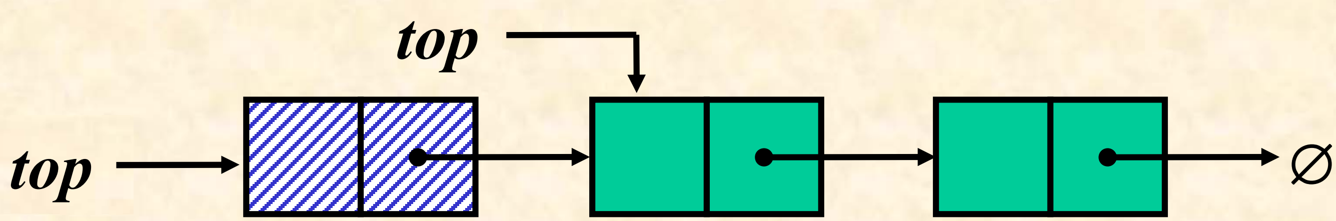
```
stack_init() {
    top = null
}
```

```
empty() {
    return top == null
}
```

```
push(val) {
    temp = new node
    temp.data = val
    temp.next = top
    top = temp
}
```



```
top() {
    if empty( ) then
        throw EmptyStackException
    else
        return top.data
}

pop() {
    if empty( ) then
        throw EmptyStackException
    else
        top = top.next
        return
}
```



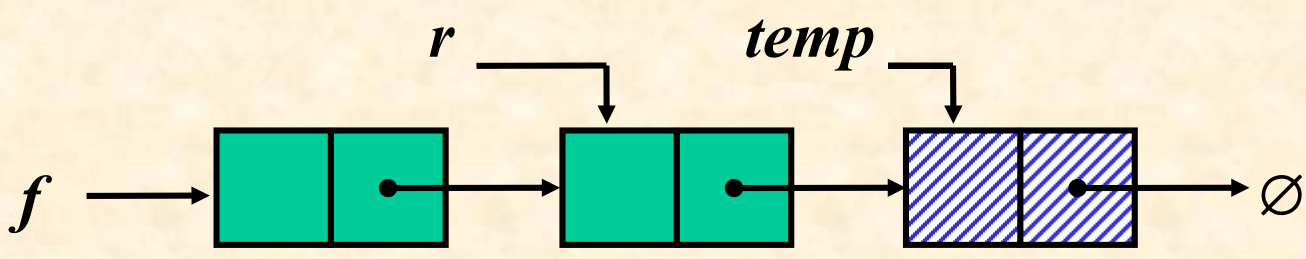
Example: Implementing Queue using Linked List

- Two pointers, *f* and *r*, are needed to reference the front and rear nodes of a queue. A new node is added to the rear of the queue.
- An empty queue has *f* = *r* = *null*

```
queue_init() {
    f = null
    r = null
}
```

```
empty() {
    return f == null
}
```

```
enqueue(val) {
    temp = new node
    temp.data = val
    temp.next = null
    if (empty())
        r = f = temp
    else {
        r.next = temp
        r = temp
    }
}
```

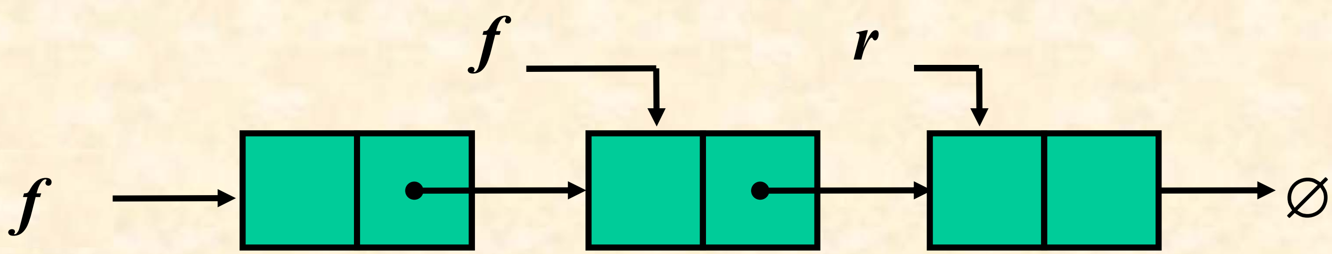


Note: the alogs. for *dequeue* and *front* will be left as exercise.

Algorithms for Queue functions (cont.)

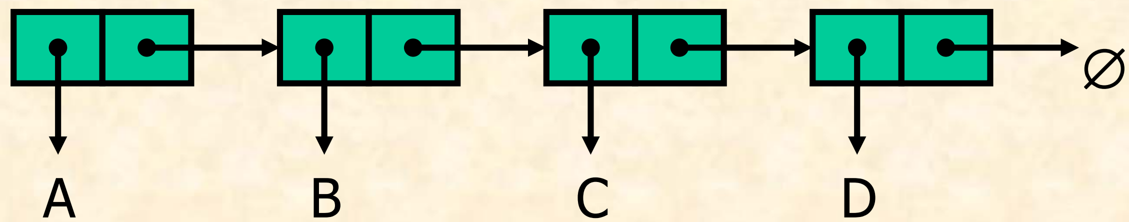
```
dequeue() {
    if (empty())
        throw QEmptyException
    else {
        //does queue contain one item?
        if (r == f)
            r = f = null
        else {
            f = f.next
        }
    }
}
```

```
front() {
    if (empty())
        throw QEmptyException
    else return  f.data
}
```

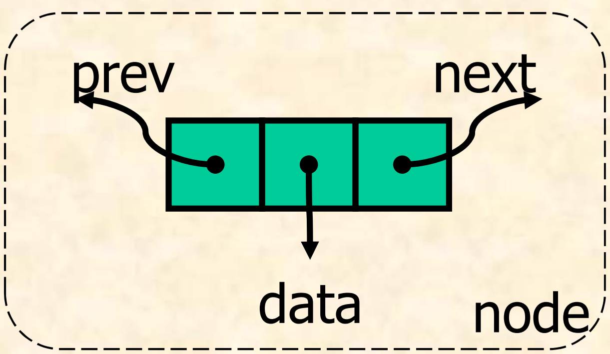


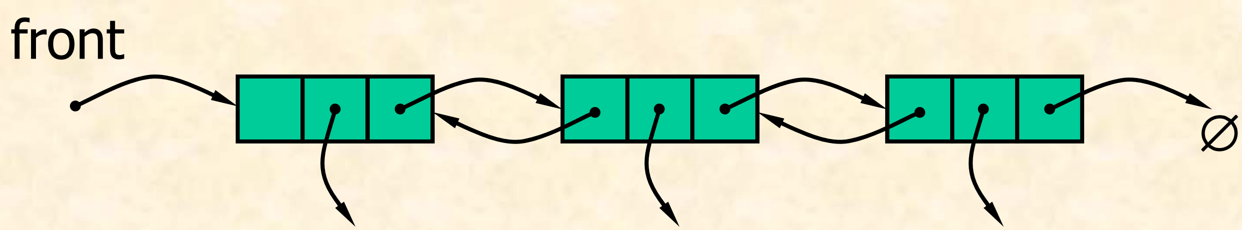
Doubly Linked Lists

- Singly Linked List: the *next* field can be used to reference the following next element
  - However, it is difficult to make reference to the previous element.



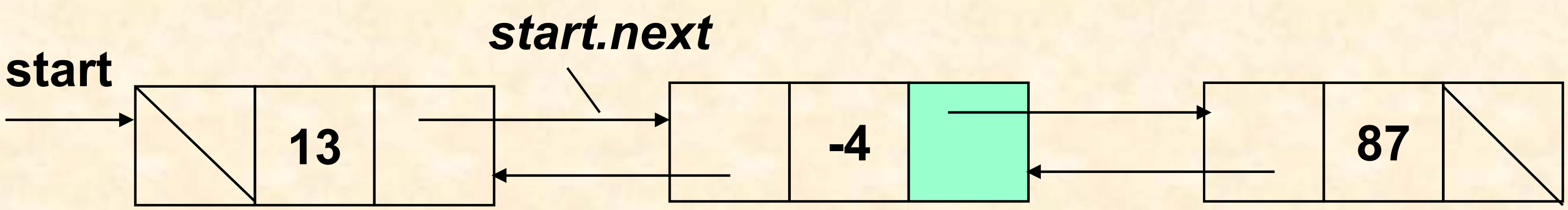
- Doubly Linked List: each node, except the first node, has a *prev* field that references the previous node in the list.





Example

- ✓ ***start.next points to the node that contains -4***
- ✓ ***start.next.next points to the node that has data 87***

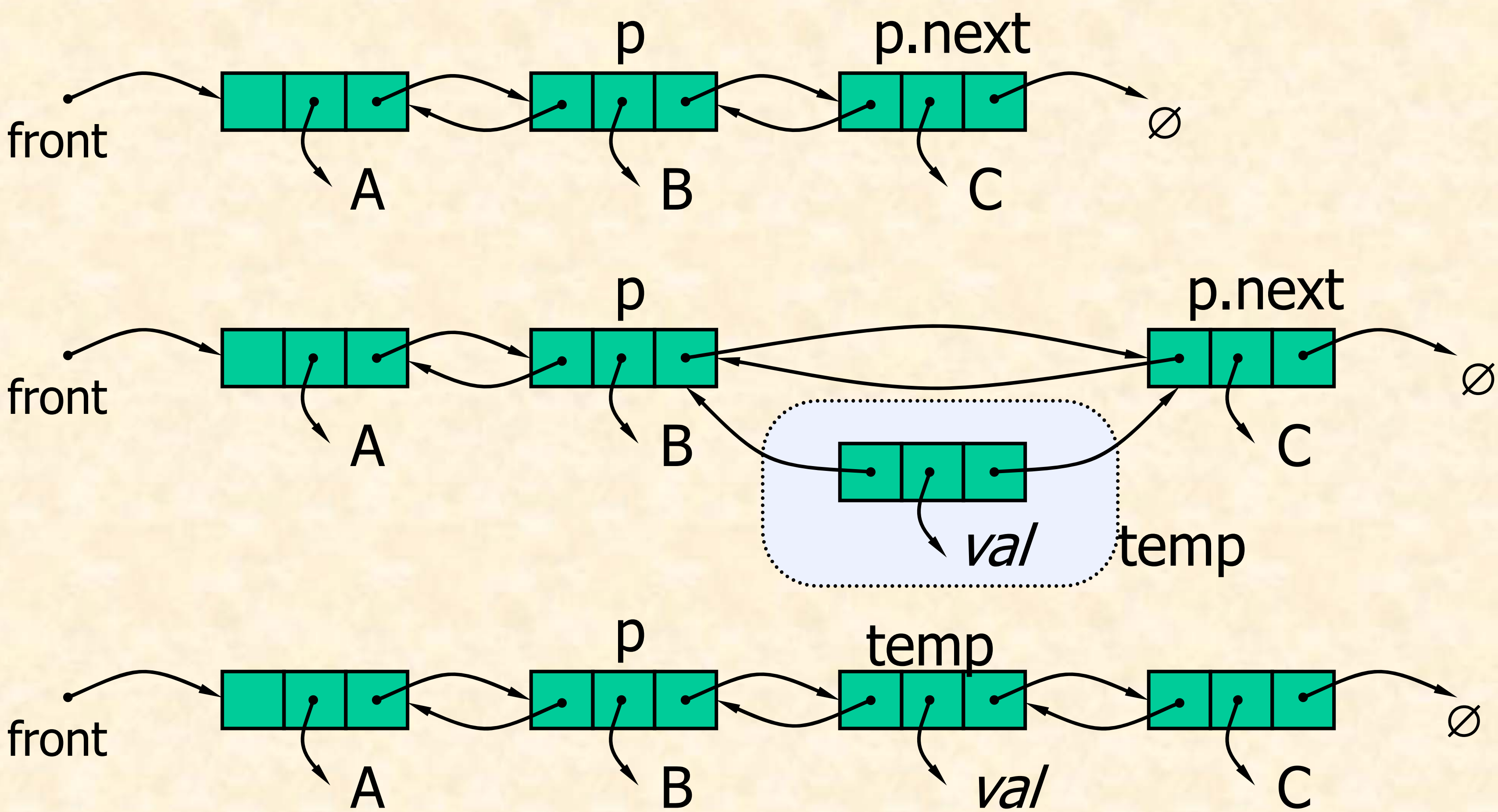


- ✓ ***start.next.prev points to node with value 13***

start.next.next.data = 87 **start.next.next.prev.data = -4**

Insertion

- insertAfter(*p*, *val*): insert a new node with value *val* after the node referenced by *p*



Insertion Algorithm

***insertAfter*(*p*,*val*) {**

**   *temp* = new *node***

**   *temp.data* = *val***

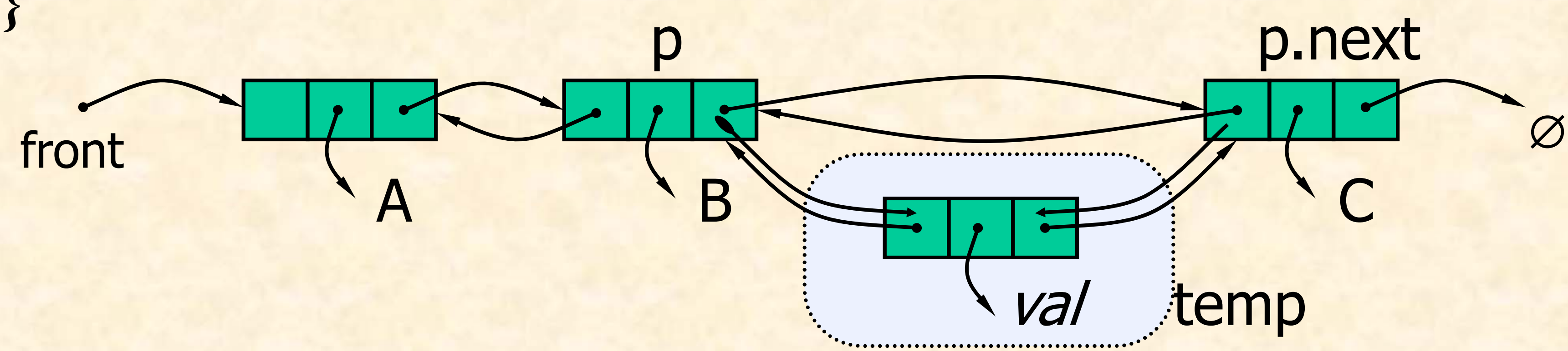
**   *temp.next* = *p.next*** // link *temp* to *p*'s successor

**   *temp.prev* = *p*** // link *temp* to *p*'s predecessor

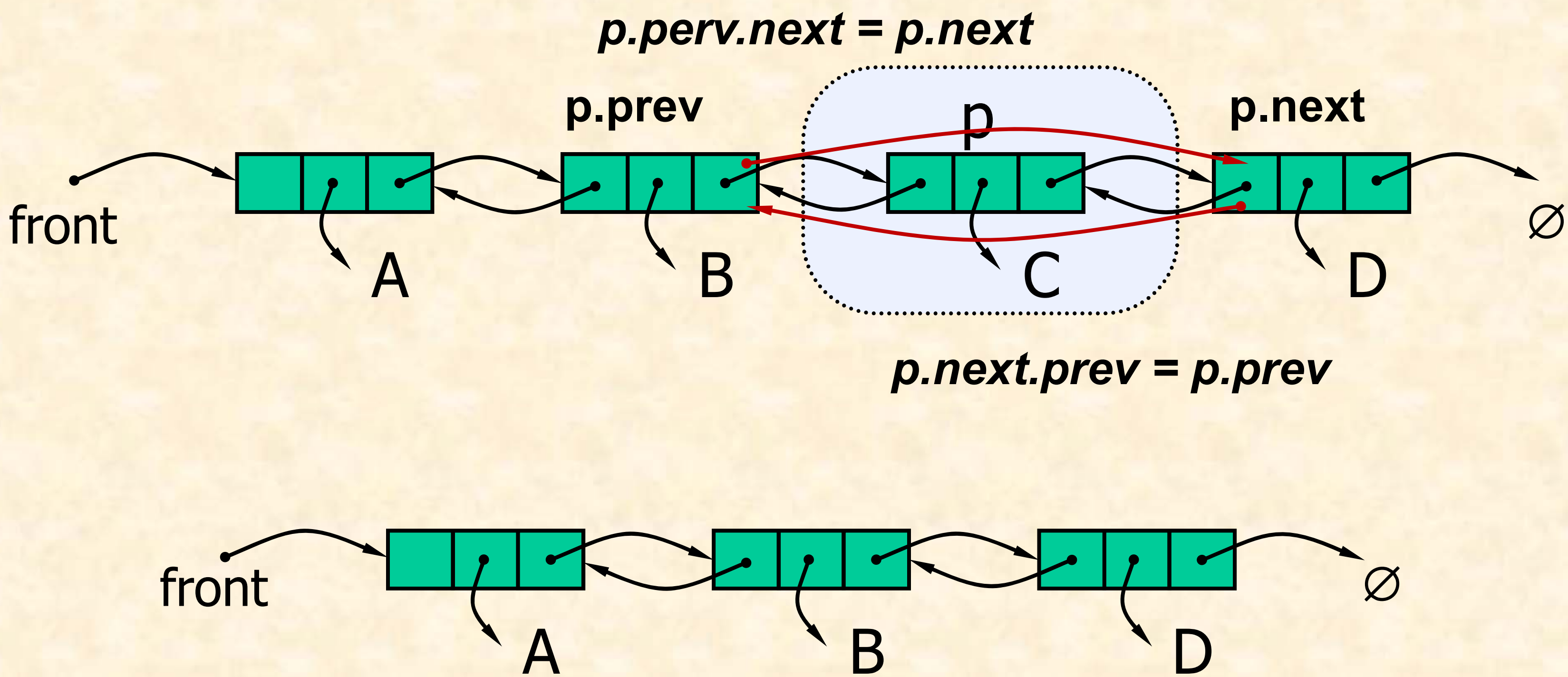
**   *p.next.prev* = *temp*** // link *p*'s old successor to *temp*

**   *p.next* = *temp*** // link *p* to its new successor, *temp*

}



Deletion



p.perv.next = p.next
p.prev
p
p.next
front
A
B
C
D
∅
p.next.prev = p.prev
front
A
B
D
∅

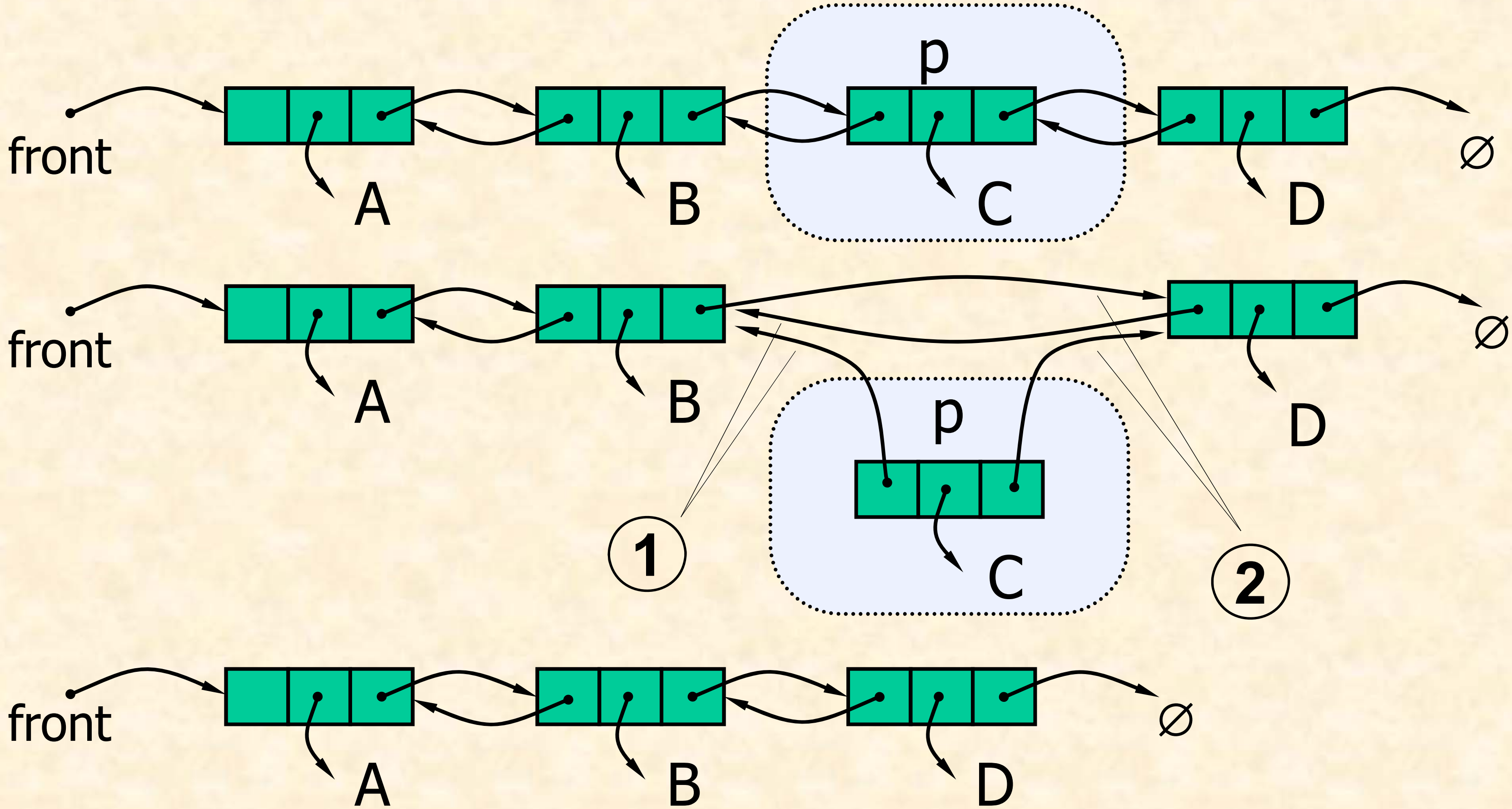
Deletion

Delete(p): delete a node referenced by *p* {

p.next.prev = p.prev 1

p.perv.next = p.next 2

}



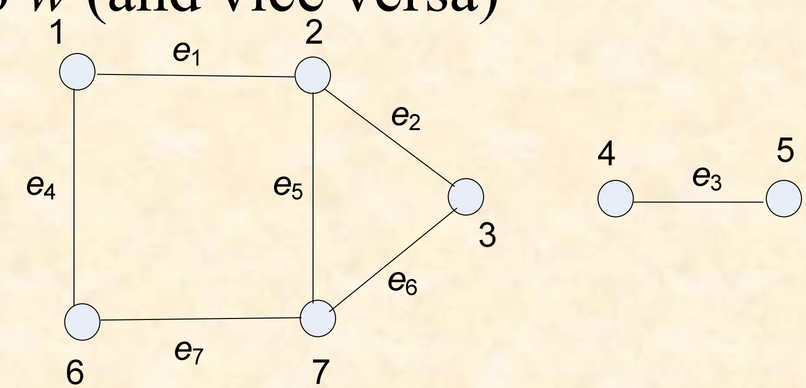
Graphs

- Graph can be used to represent cities connected by roads, networking equipments connected by transmission links, etc.
- A graph $G = (V,E)$ consists of a set V of vertices (or nodes) and a set E of edges such that each edge $e \in E$ is associated with a pair of vertices.
  - For an edge e associated with the vertices v and w, we write $e=(v, w)$ or $e=(w, v)$
    - we say that v is ***adjacent*** to w (and vice versa)
- Example: $G = (V,E)$

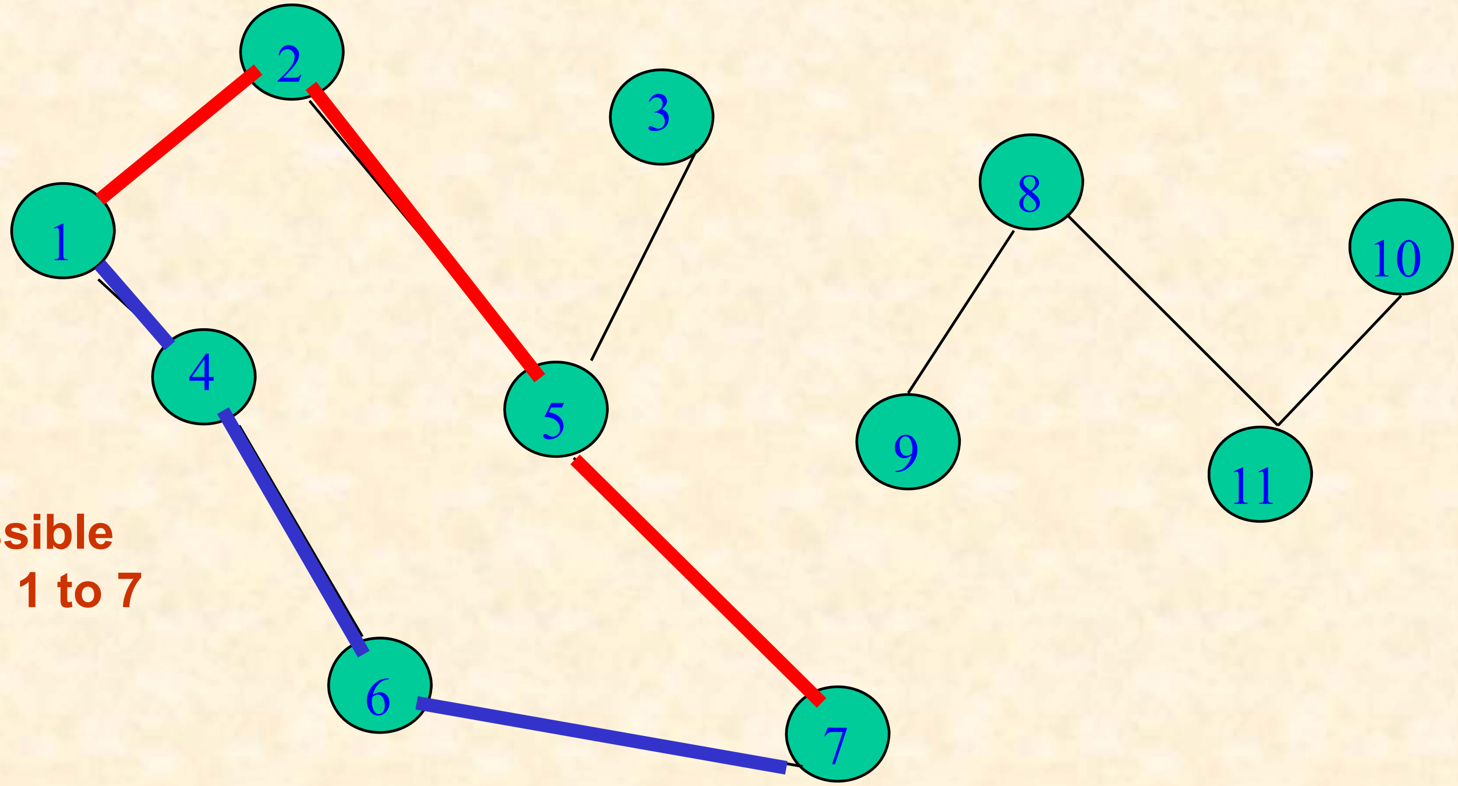
  $V = \{1,2,3,4,5,6,7\}$,

  $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$

  $e_1=(1,2)$, $e_2=(2,3)$, etc.



Applications—Communication Network



Find all possible routes from 1 to 7

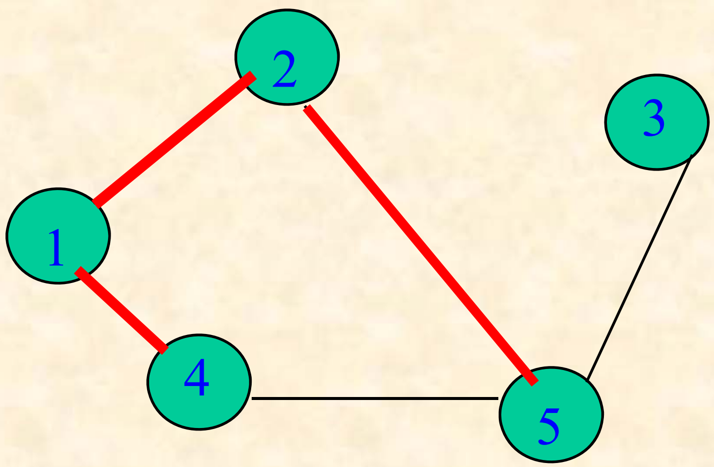
- Vertex = city, edge = communication link.

Graph Representation

- How to represent a graph?
  - Use data structure
- Graph Data Structures
  - Adjacency Matrix
  - Adjacency Lists

Adjacency Matrix

- 0/1 n x n matrix, where n = # of vertices
- A(i,j) = 1 iff (i,j) is an edge



	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

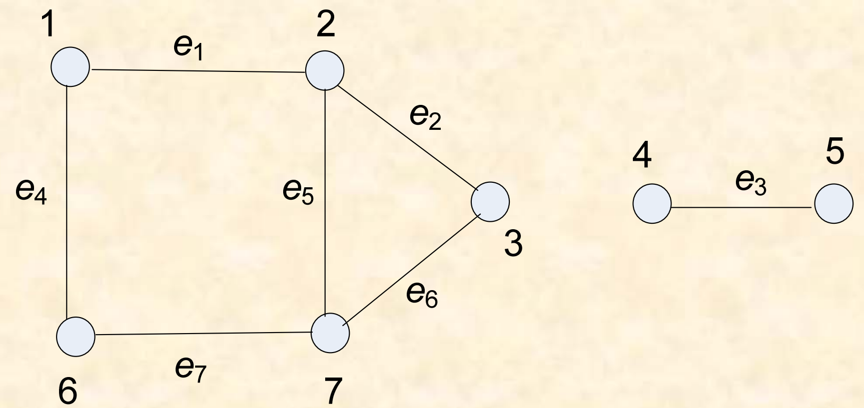
- How should a graph be represented?
- One way: ***adjacency matrix***
  - The rows and columns of the matrix represent the vertices
  - The entry on the matrix in row i and column j

    = 1 if (i,j) is an edge

    = 0 if (i,j) is not an edge

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
6 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
7 & 0 & 1 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

The degree of *v* is the number of edges incident on *v*.



- The degree of *v* is the number of edges incident on *v*.
- Algorithm: To find the degree of each vertex in a graph

  Input: an adjacency matrix *am*

```
degrees(am) {
   for i = 1 to am.last {
         count = 0
         for j =1 to am.last
                if (am[i][j] = = 1)
                      count = count + 1
         println ("vertex ", i, "has degree ", count)
   }
}
```

Note: For a graph with *n* vertices, the above algo. runs in time $O(n^2)$

Computing Vertex Degrees Using an Adjacency Matrix

Algorithm Adjacency Matrix

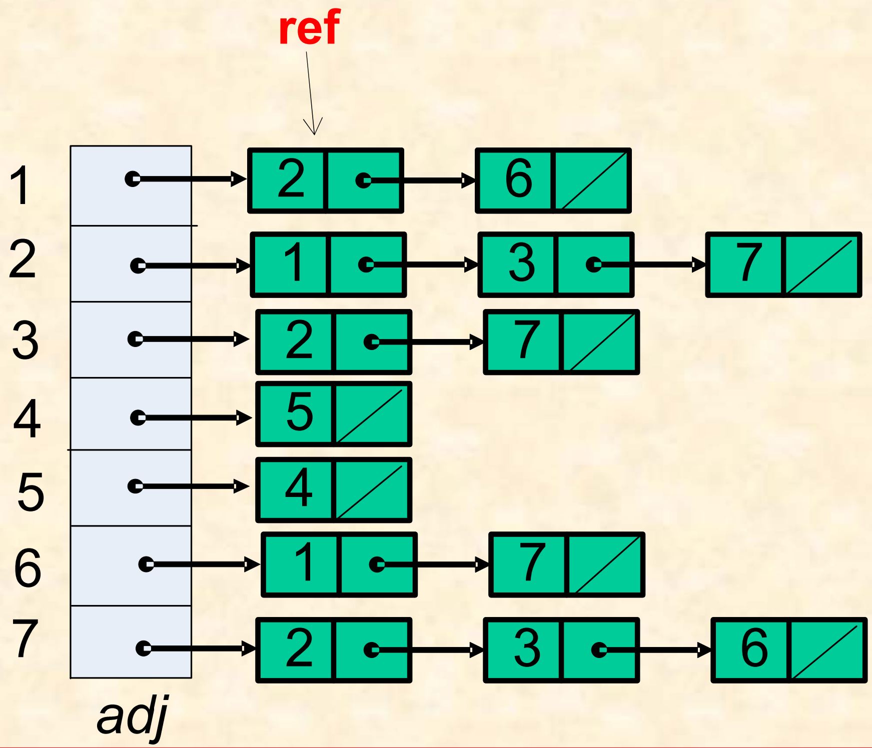
```
degrees2(am) {
    for i = 1 to am.last {
        count = 0
        for j = 1 to am.last
            if (am[i][j] == 1)
                count = count + 1
        println("vertex " + i + " has degree " + count)
    }
}
```

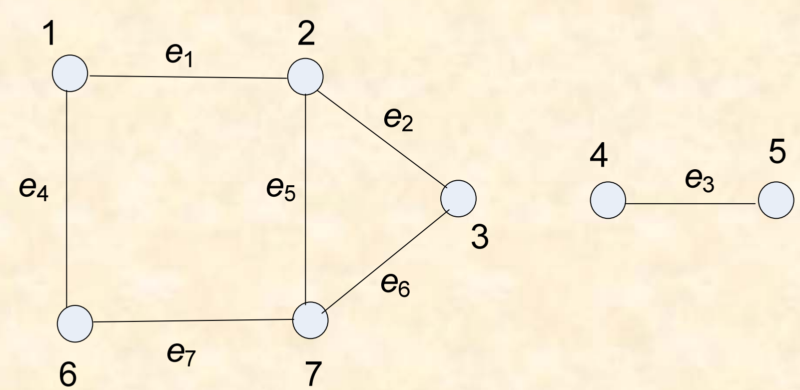
		1	2	3	4	5
i →	1	0	1	0	1	0
i →	2	1	0	0	0	1
i →	3	0	0	0	0	1
i →	4	1	0	0	0	1
i →	5	0	1	1	1	0

Time complexity = $O(n^2)$

- A more common way to represent a graph using linked list: ***adjacency lists***.
- An array, *adj*, is used to access the linked lists
  - *adj*[*i*] is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex i

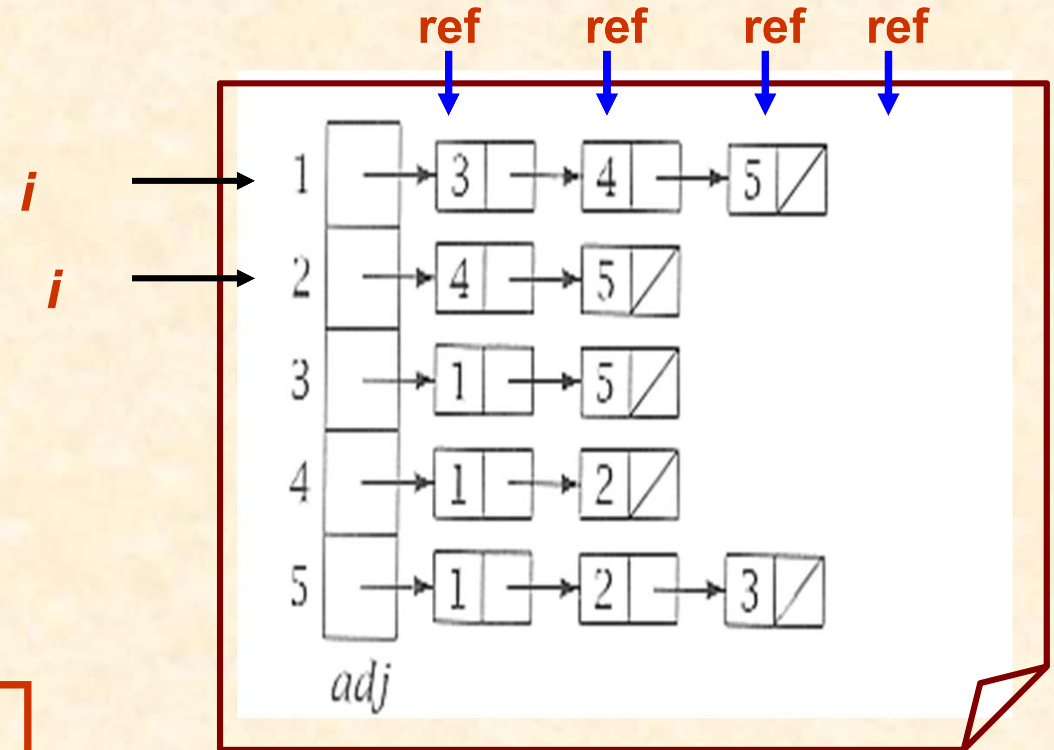
ref = adj[1]





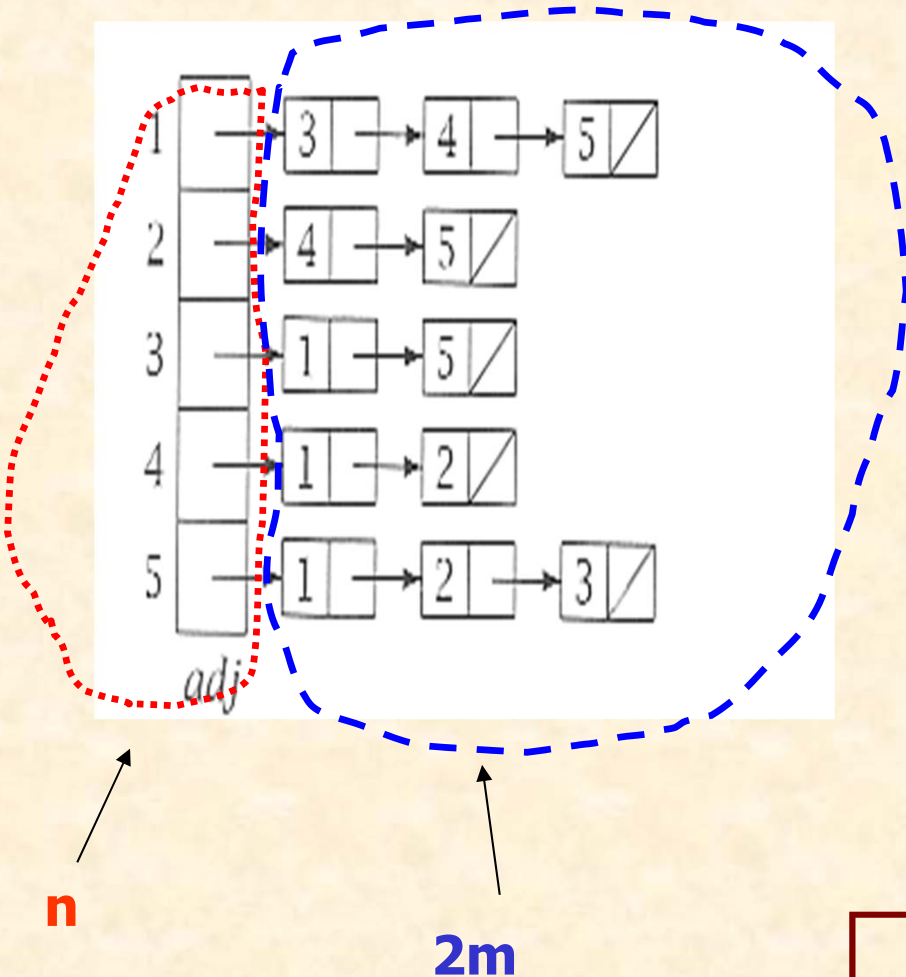
Computing Vertex Degrees Using Adjacency Lists

Algorithm Adjacency Lists




```
degrees1(adj) {
  for i = 1 to adj.last {
      count = 0
      ref = adj[i]
      while (ref != null) {
          count = count + 1
          ref = ref.next
      }
      println("vertex " + i + " has degree " + count)
  }
}
```

Computing Vertex Degrees Using Adjacency Lists



- Let m = number of edges in the graph
- Number of adjacency lists = n
- Each edge (i,j) is represented twice in the adj lists: j appears once in i's list and i appears once in j's list
- Hence there is a total of 2m nodes in the adjacency lists

Time complexity = O(n+m)

- Algorithm: To find the degree of each vertex in a graph
  Input: an adjacency list *adj*

```
degrees(adj) {
   for i = 1 to adj.last {
         count = 0
         ref = adj[i]
         while (ref != null) {
                count = count + 1
                ref = ref.next
         }
         println ("vertex ", i, "has degree ", count)
   }
}
```

Note: For a graph with n vertices and m edges, the above algo. runs in time $O(m+n)$.

- an edge is represented twice in the adjacency list, e.g. an edge connecting vertices i and j, the edge will appear on both i's and j's list
- The while loop will runs for $2m$ times

Example of creating a linked list of integers

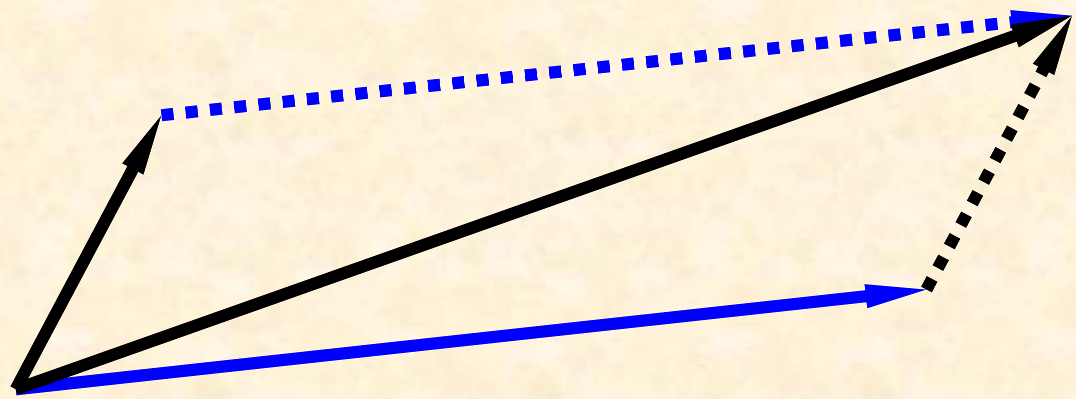
```
struct num_list {
    int num;
    struct num_list *next;
};

void main()
{
    num_list* head = new num_list;
    head->num=1;
    head->next=NULL;
    last = head;
    for (i=2; i<=10; i++)
    {
        num_list* temp = new num_list;
        last->next=temp;
        temp->num=i;
        temp->next=NULL;
        last = temp;
    }
}
```

back1
back2

```
#include <stdio.h>

int i;
struct num_list {
     int num;
     struct num_list *next;
};

struct num_list *head,*temp,*last;

/* A simple program that stores 10
     numbers in a linked list and then
     prints out the list */

void main()
{
     num_list* head = new num_list;
     head->num=1;
     head->next=NULL;

     last = head;
     for (i=2; i<=10; i++)
     {
          num_list* temp = new num_list;
          last->next=temp;
          temp->num=i;
          temp->next=NULL;
          last = temp;
     }

     last=head;
     while (last != NULL)
     {
          printf("\n %d ",last->num);
          last = last->next;
     }

}
```

back1
back2

Vectors



Vectors

- Suppose we are given a linear sequence *S* that contains *n* elements.
- We uniquely refer to each element in *e* in *S* using an integer in the range [0, *n*-1] that is equal to the number of elements of *S* that precede *e* in *S*.
- The rank of an element *e* in *S* is defined to be the number of elements that are before *e* in *S*
  - Hence the first element in S has rank 0
  - The last element in S has rank *n*-1

	u	w	z		
rank	0	1	2	3	4

Vectors (contd)

- Note that the rank of an element may change whenever the sequence is updated
  - Eg: if we insert a new element at the beginning of the sequence, the rank of each of the other elements increases by one
- A Vector : is a linear sequence that supports access to its elements by their ranks.

Insert "d" at rank 0

	u	w	z		
rank	0	1	2	3	4

The Vector ADT

- The Vector ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it)
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

Rank Operations

- Operations based on rank

Insert “3” at rank 0

3				
0	1	2	3	4

Insert “8” at rank 0

8	3			
0	1	2	3	4

Insert “11” at rank 1

8	11	3		
0	1	2	3	4

Rank Operations

- Operations based on rank

Access element at rank 0

8	3	5		
0	1	2	3	4

Remove element at rank 1

8	5			
0	1	2	3	4

Access element at rank 2

8	5			
0	1	2	3	4

error

The Vector ADT

- Main vector operations:
  - elemAtRank(int r): returns the element at rank r without removing it
  - replaceAtRank(int r, object o): replace the element at rank r with o and return the old element
  - insertAtRank(integer r, object o): insert a new element o to have rank r
  - removeAtRank(integer r): removes and returns the element at rank r

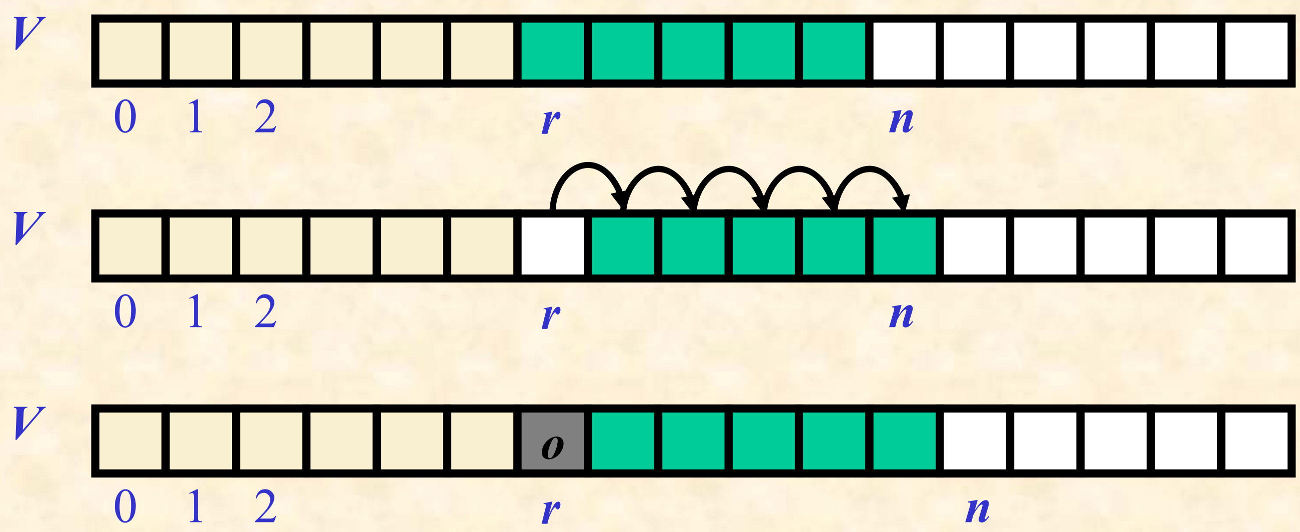
Array-based Vector

- Use an array ***V*** of size ***N***
- A variable ***n*** keeps track of the size of the vector (number of elements stored)
- Operation ***elemAtRank***(***r***) is implemented in ***O***(1) time by returning ***V***[***r***]

Insertion

- In operation ***insertAtRank***(***r***, ***o***), we need to make room for the new element by shifting forward the ***n*** – ***r*** elements ***V***[***r***], …, ***V***[***n*** – 1]
- In the worst case (***r*** = 0), this takes ***O***(***n***) time

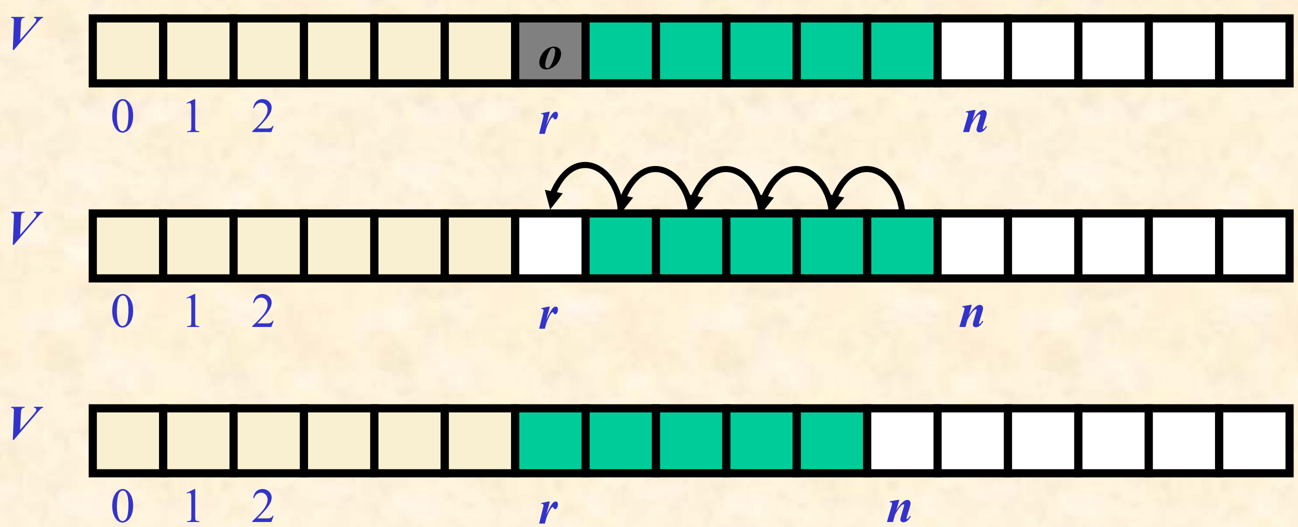
```
Algorithm insertAtRank(r, o)

    for i = n-1 downto r do
        A[i+1] = A[i]

    A[r] = "o"
    n = n + 1
```



Deletion

- In operation ***removeAtRank***(***r***), we need to fill the hole left by the removed element by shifting backward the $\boldsymbol{n - r - 1}$ elements $\boldsymbol{V}[\boldsymbol{r} + 1], \ldots, \boldsymbol{V}[\boldsymbol{n} - 1]$
- In the worst case ($\boldsymbol{r} = 0$), this takes $\boldsymbol{O}(\boldsymbol{n})$ time

```
Algorithm removetAtRank( r)

  e = A[r]

  for i = r  to  n-2 do
     A[i] = A[i+1]

  n = n – 1
  return e
```



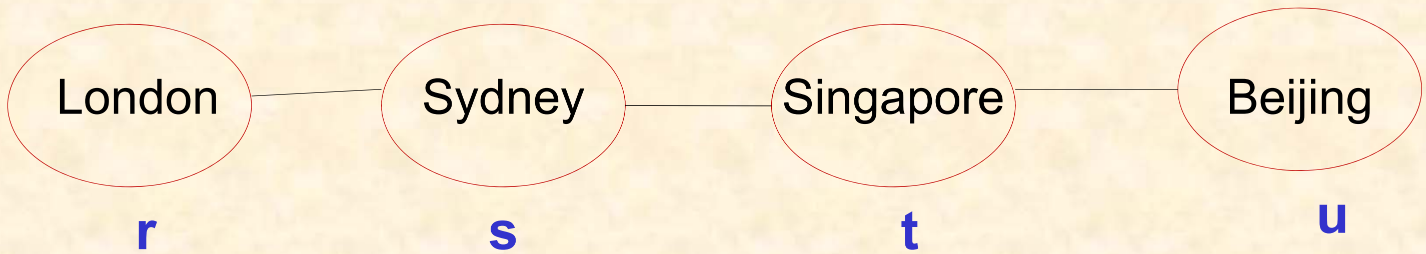
Performance

- In the array based implementation of a Vector
  - The space used by the data structure is $\boldsymbol{O}(\boldsymbol{n})$
  - ***elemAtRank*** and ***replaceAtRank*** run in $\boldsymbol{O}(1)$ time
  - ***insertAtRank*** and ***removeAtRank*** run in $\boldsymbol{O}(\boldsymbol{n})$ time
- In an ***insertAtRank*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

LISTS

Lists

- Extends a linked list to store items based on a position in the list
- Position
  - denotes the object location in a list
  - are defined relatively (in terms of neighbors)
    - Eg: position **t** is after position **s** and before position **u**



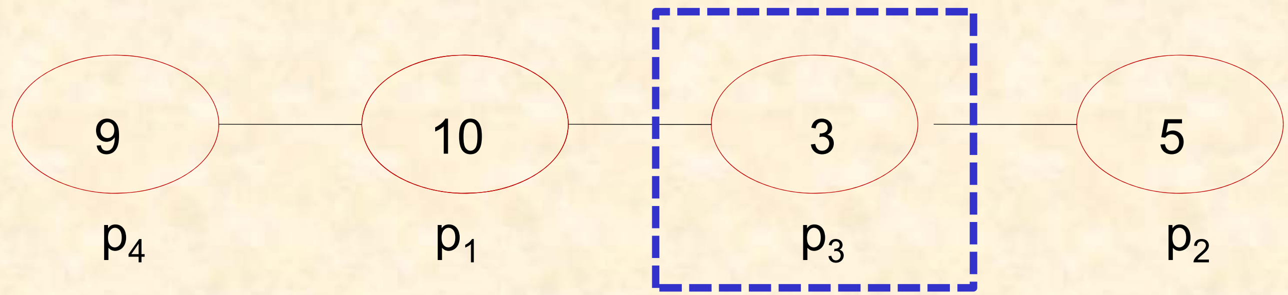
List ADT Methods

- Query methods:

  – isFirst(*p*) : return a Boolean value indicating whether the given position *p* is the **first one** in the list

  - isLast(*p*) : return a Boolean value indicating whether the given position *p* is the **last one** in the list

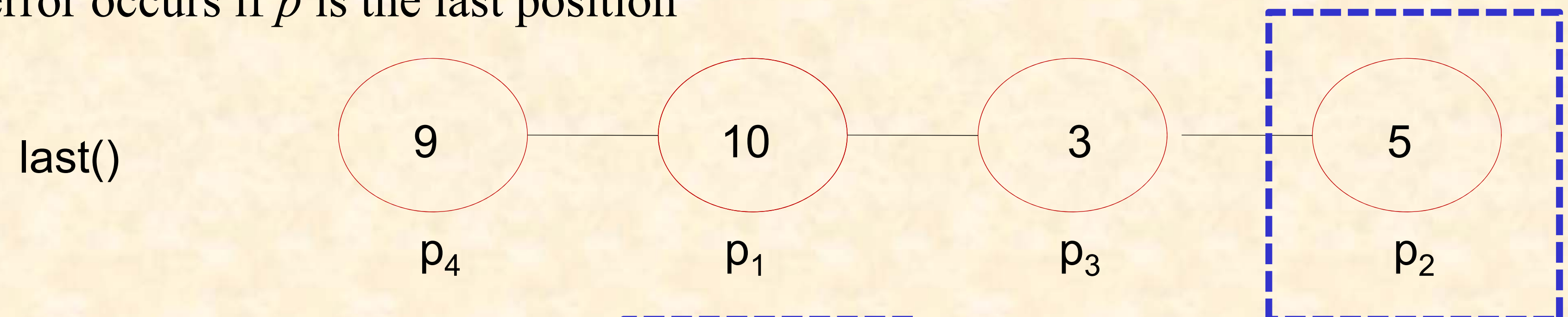
isFirst(p_3)

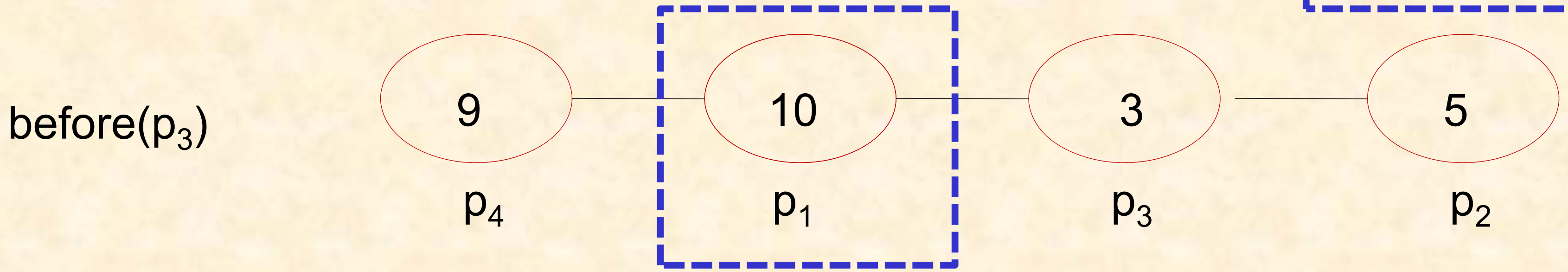


List ADT Methods (contd)

Accessor methods:

- first() : return the position of the first element in the list; an error occurs if list is empty
- last() : return the position of the last element in the list; an error occurs if list is empty
- before(p) : return the position of the element preceding the one at position p in the list; an error occurs if p is the first position
- after(p) return the position of the element in the list following the one at position p; an error occurs if p is the last position



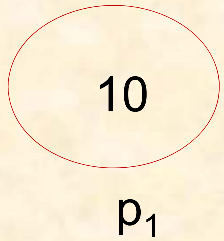


List ADT Methods (contd)

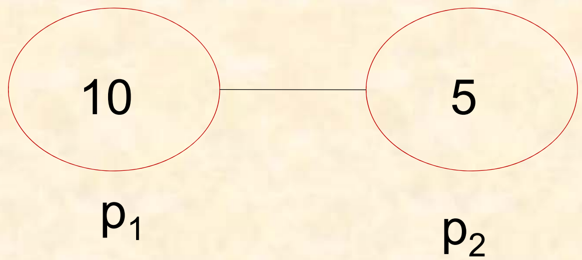
- Update methods:
  - replaceElement(p, e) : replace the element at position p with e, returning the element formerly at position p
  - swapElements(p, q) : swap the elements at positions p and q, so that the element that is a position p moves to position q and the element that is a position q moves to position p
  - insertBefore(p, e) : insert a new element e before position p in the list
  - insertAfter(p, e) : insert a new element e after position p in the list
  - insertFirst(e) : insert a new element e into the list as the first element
  - insertLast(e) : insert a new element e into the list as the last element
  - remove(p) : remove the element at position p in the list

Operations

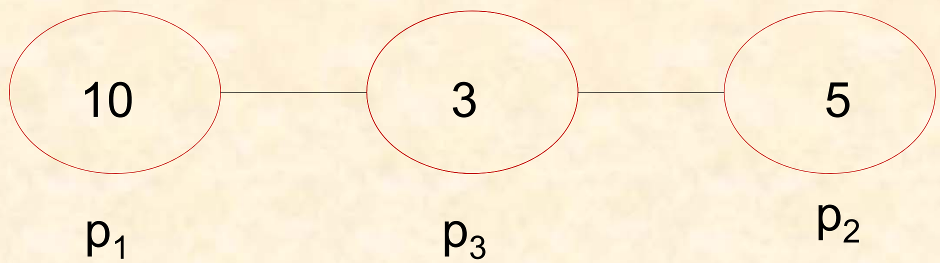
insertFirst(10)



insertAfter(p_1,5)

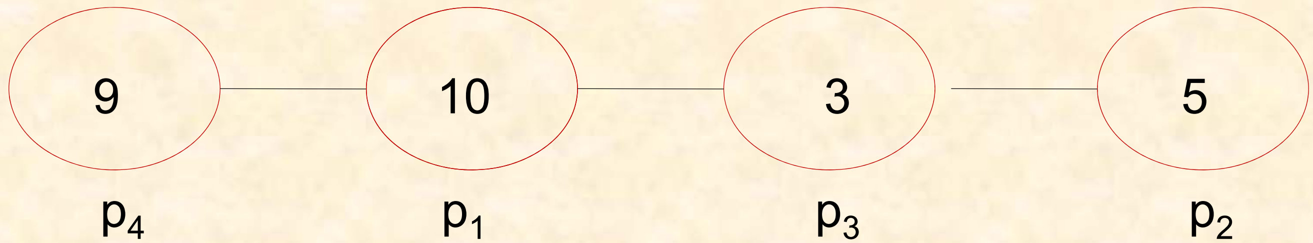


insertBefore(p_2,3)

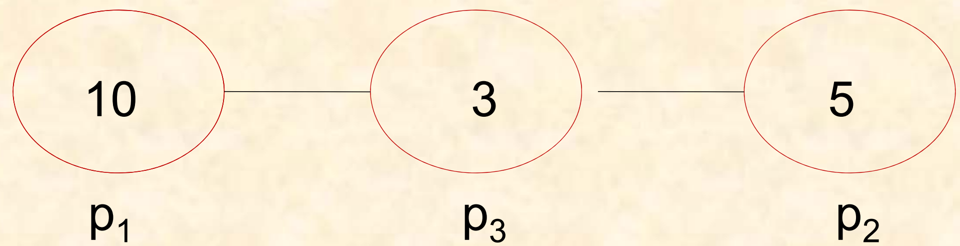


Operations

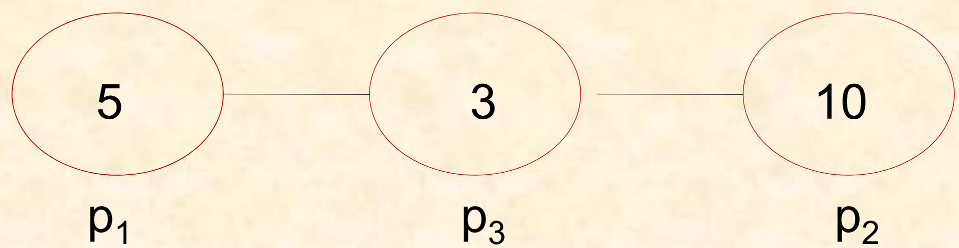
insertFirst(9)



remove(p_4)

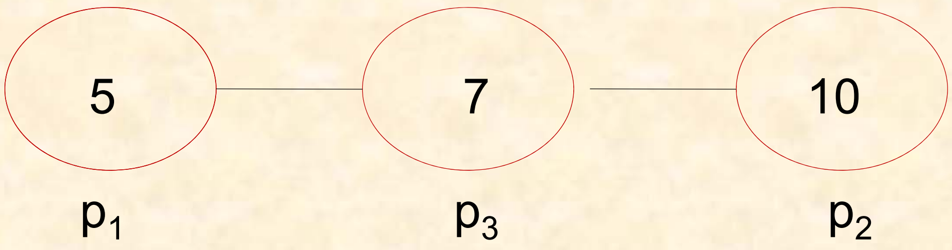


swapElements(p_1,p_2)

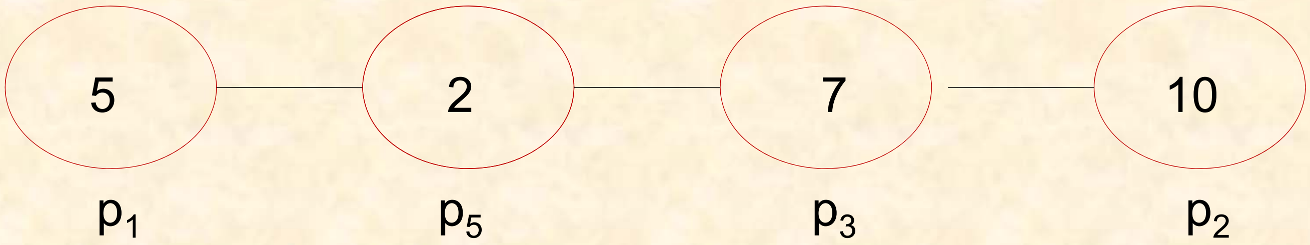


Operations

replaceElement(p_3,7)

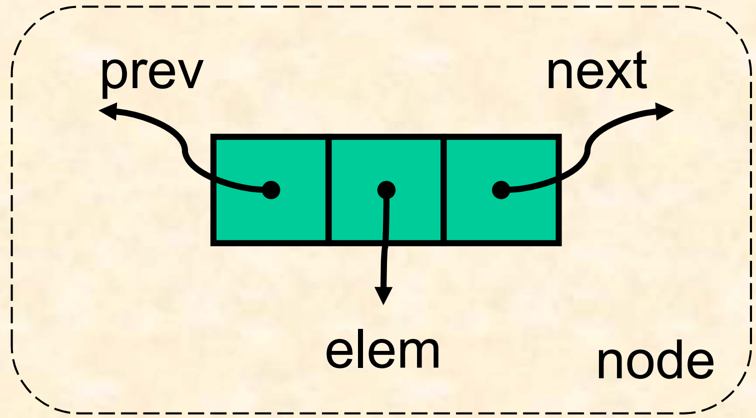


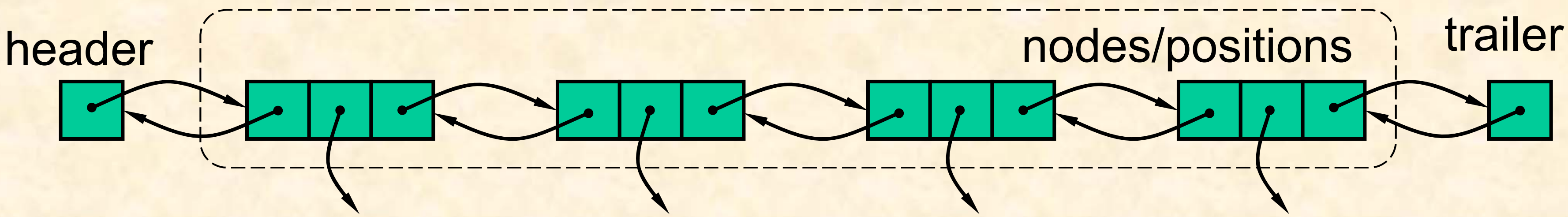
insertAfter(first(),2)



Implementations of LIST ADT Methods

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
  - header node has a valid next reference but a null prev reference
  - Trailer node has a valid prev reference but a null next reference

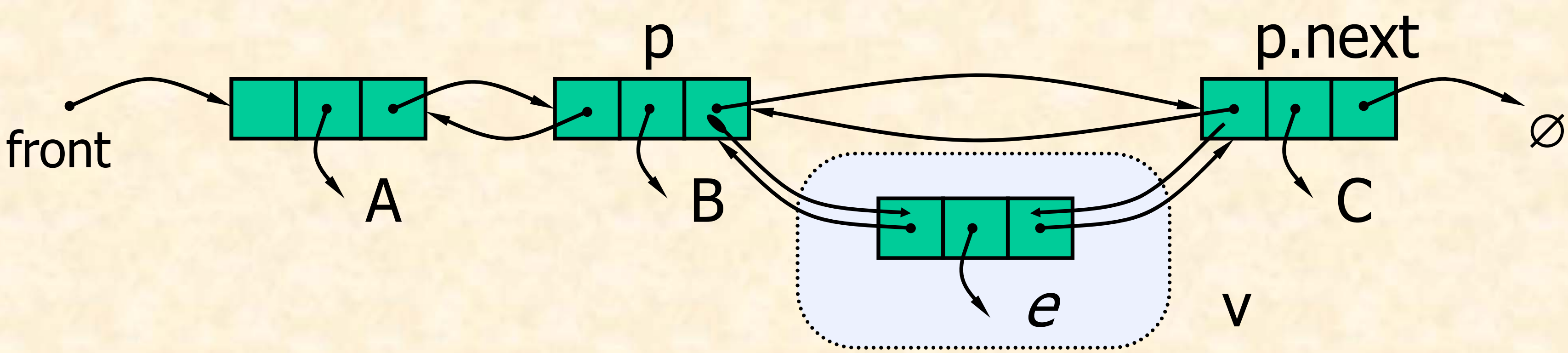




Implementations of LIST ADT Methods

Algorithm insertAfter (*p*,*e*)

v = new node
v.data = e
v.prev = p
v.next = p.next
(p.next).prev = v
p.next = v
return v



Implementations of LIST ADT Methods (contd)

Algorithm remove (p)

t = *p.data*

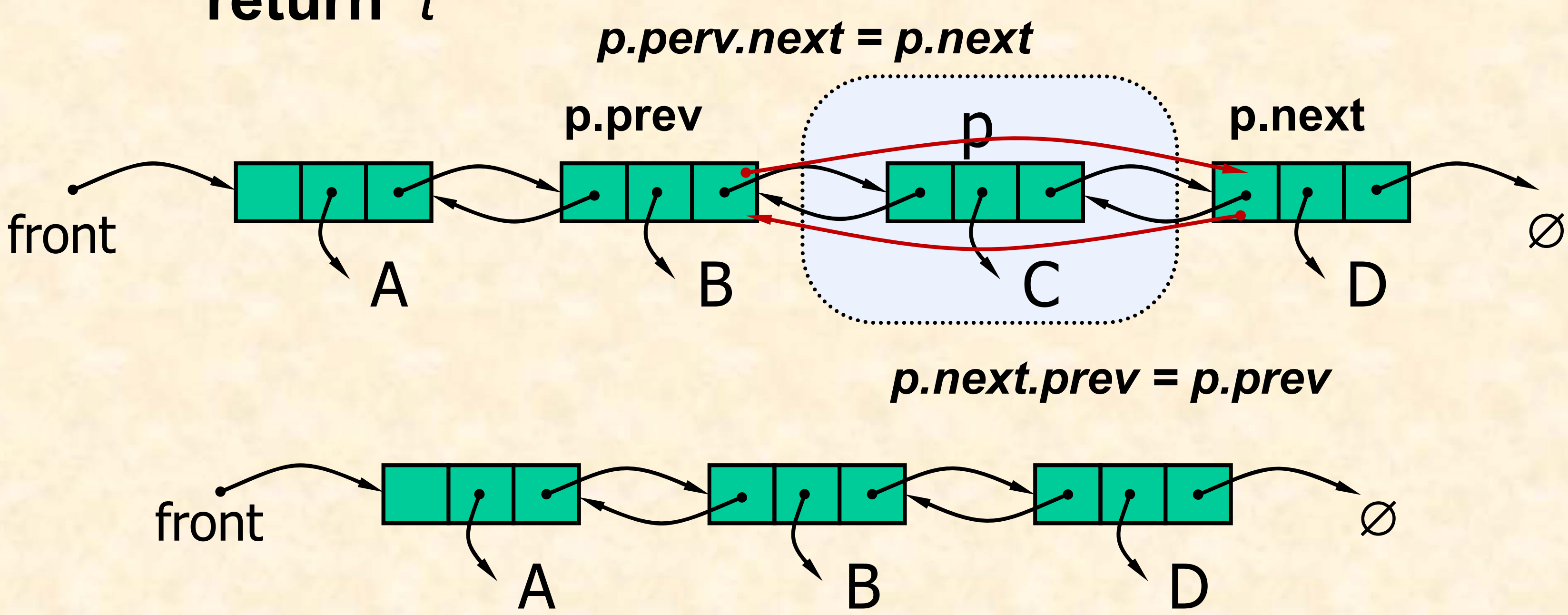
p.next.prev = *p.prev*

p.prev.next = *p.next*

p.prev = *null*

p.next = *null*

return *t*



Priority Queues ADT

Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Applications:
  - Auctions
  - Stock market

The Priority Queue ADT

- A prioriy queue P supports the following methods:

-size(): Return the number of elements in P

-isEmpty(): Test whether P is empty

-insertItem(k,e): Insert a new element e with key k into P

-minElement(): Return (but don't remove) an element of P with smallest key; an error occurs if P is empty.

-minKey(): Return the smallest key in P; an error occurs if P is empty

-removeMin(): Remove from P and return an element with the smallest key; an error condidtion occurs if P is empty.

Keys and Total Order Relations

- A Priority Queue ranks its elements by **key** with a **total order** relation
- Keys: Every element has its own key

  Keys are not necessarily unique
- Total Order Relation, denoted by $\leq$

  Reflexive: $k \leq k$

  Antisymetric: if $k1 \leq k2$ and $k2 \leq k1$, then $k1 = k2$

  Transitive: if $k1 \leq k2$ and $k2 \leq k3$, then $k1 \leq k3$

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

Comparators ADT (contd)

- Methods of the Comparator ADT, all with Boolean return type
  - isLessThan(a, b) : True if and only if $a < b$
  - isLessThanOrEqualTo(a,b) : True if and only if $a \leq b$
  - isEqualTo(a,b) : True if and only if $a = b$
  - isGreaterThan(a, b) : True if and only if $a > b$
  - isGreaterThanOrEqualTo(a,b) : True if and only if $a \geq b$
  - isComparable(a) : True if and only if a can be compared

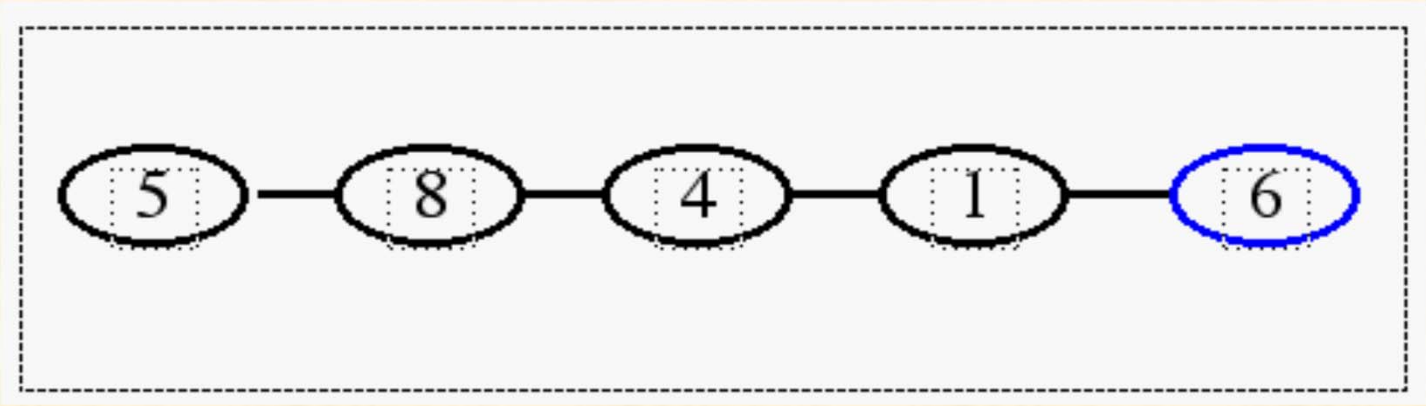
Sorting with a Priority Queue

- A Priority Queue P can be used for sorting a sequence S by:
  - ***inserting*** the elements of S into P with a series of insertItem(e, e) operations
  - ***removing*** the elements from P in increasing order and putting them back into S with a series of removeMin() operations

```
Algorithm PriorityQueueSort(S, P):
        Input: A sequence S storing n elements, on which a
                total order relation is defined, and a Priority Queue
                P that compares keys with the same relation
        Output: The Sequence S sorted by the total order relation
        while !S.isEmpty() do
                e ← S.removeFirst()
                P.insertItem(e, e)
        while P is not empty do
                e ← P.removeMin()
                S.insertLast(e)
```

Implementation with an Unsorted Array

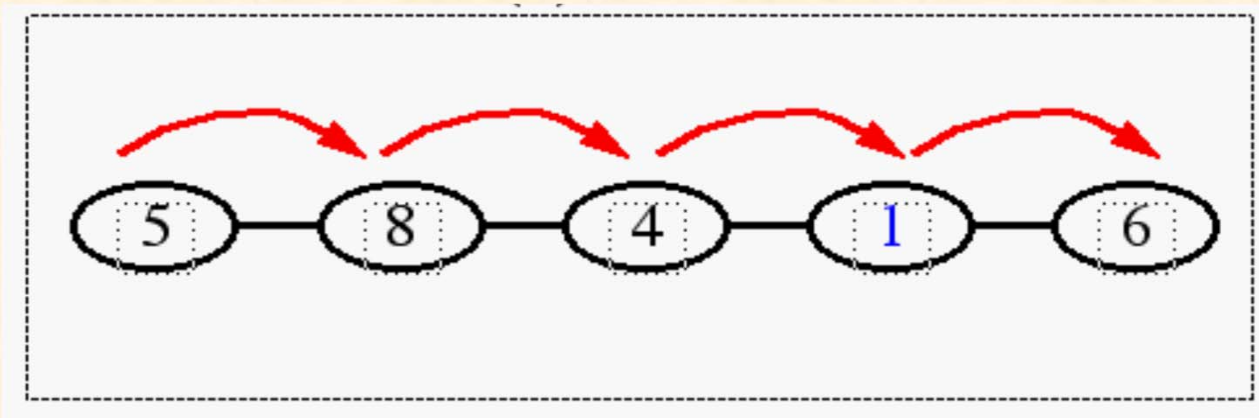
- Let's try to implement a priority queue with an unsorted array S.
- The elements of S are a composition of two elements, k, the key, and e, the element.
- We can implement insertItem() by using insertLast() on the array. This takes ***O***(1) time.



•However, because we always insert at the end, irrespectively of the key value, our sequence is not ordered.

Implementation with an Unsorted Array (contd.)

- Thus, for methods such as minElement(), minKey(), and removeMin(), we need to look at all the elements of S. The worst case time complexity for these methods is O(n).



•Performance summary

insertItem	$O(1)$
minKey, minElement	$O(n)$
removeMin	$O(n)$

```
Algorithm PriorityQueueSort(S, P):
        Input: A sequence S storing n elements, on which a
                total order relation is defined, and a Priority Queue
                P that compares keys with the same relation
        Output: The Sequence S sorted by the total order relation
        while !S.isEmpty() do
                e ← S.removeFirst()
                P.insertItem(e, e)
        while P is not empty do
                e ← P.removeMin()
                S.insertLast(e)
```

Selection Sort

- Selection Sort is a variation of PriorityQueueSort that uses an unsorted array to implement the priority queue P.
  - Phase 1, the insertion of an item into P takes O(1) time
  - Phase 2, removing an item from P takes time proportional to the current number of elements in P

	Sequence S	Priority Queue P
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Selection Sort (cont.)

- As you can tell, a bottleneck occurs in Phase 2. The first removeMinElement operation take O(n), the second O(n-1), etc. until the last removal takes only O(1) time.
- The total time needed for phase 2 is:

$$O(n + (n-1) + \ldots + 2 + 1) \equiv O\left(\sum_{i=1}^{n} i\right)$$

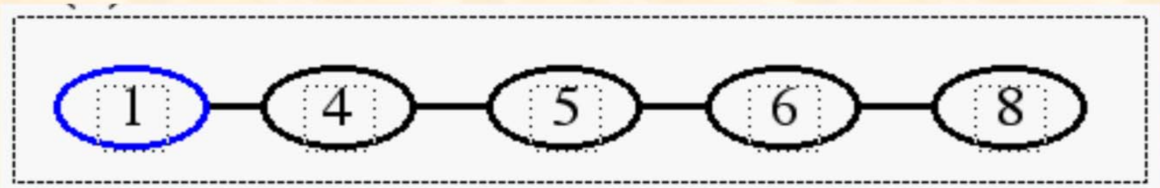
- By a well-known fact:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

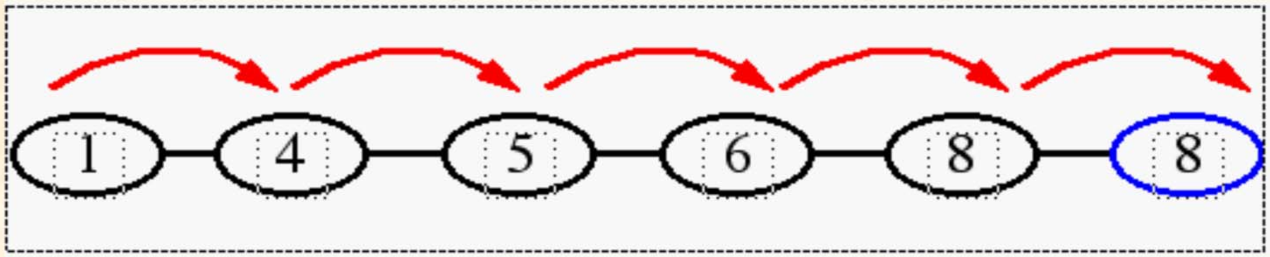
- The total time complexity of phase 2 is then $O(n^2)$. Thus, the time complexity of the algorithm is $O(n^2)$.

Implementation with Sorted Array

- Another implementation uses a array S, sorted by increasing keys
- minElement(), minKey(), and removeMin() take O(1) time



- However, to implement insertItem(), we must now scan through the entire array in the worst case. Thus insertItem() runs in O(n) time



insertItem	$O(n)$
minKey, minElement	$O(1)$
removeMin	$O(1)$

Insertion Sort

- Insertion sort is the sort that results when we perform a PriorityQueueSort implementing the priority queue with a ***sorted array***

	Sequence S	Priority Queue P
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Insertion Sort(cont.)

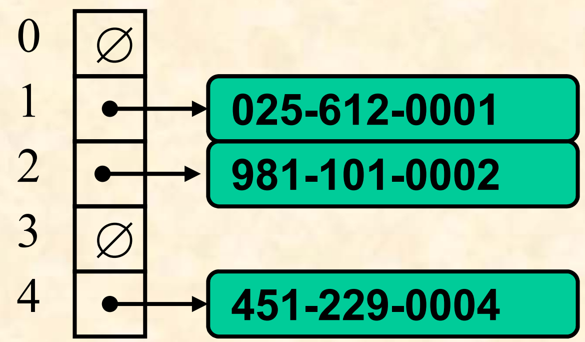
- We improve phase 2 to ***O***(n).
- However, phase 1 now becomes the bottleneck for the running time. The first **insertItem** takes ***O***(1) time, the second one ***O***(2), until the last opertation takes ***O***(n) time, for a total of ***O***(n^2) time
- Selection-sort and insertion-sort both take ***O***(n^2) time
- Selection-sort will ***alway***s executs a number of operations proportional to n^2, no matter what is the input sequence.
- The running time of insertion sort varies depending on the input sequence.
- Neither is a good sorting method, except for small sequences
- We have yet to see the ultimate priority queue....

Sequence-based Priority Queue

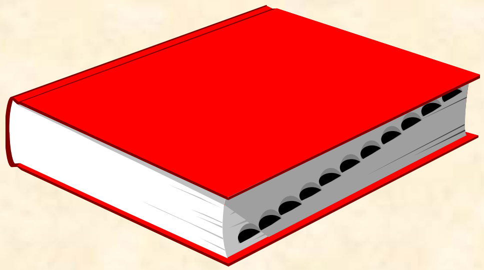
- Implementation with an unsorted sequence
  - Store the items of the priority queue in a list-based sequence, in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted sequence
  - Store the items of the priority queue in a sequence, sorted by key
- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

Dictionaries and Hash Tables



Dictionary ADT



- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

Dictionary ADT (contd)

- Dictionary ADT methods:
  - findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
  - insertItem(k, o): inserts item (k, o) into the dictionary
  - removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
  - size(), isEmpty()
  - keys() : returns keys stored in dictionary
  - Elements() : return the elements stored in dictionary

Log File

- A log file is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked lists or a circular array), in arbitrary order
- Performance:
  - insertItem takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - findElement and removeElement take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Functions and Hash Tables

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:

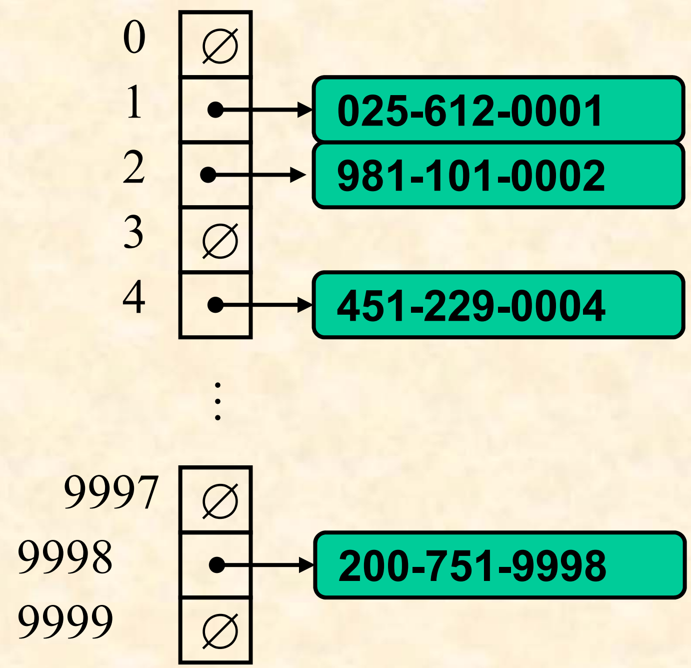
  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $\boldsymbol{h}(\boldsymbol{x})$ = last four digits of $\boldsymbol{x}$



Hash Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

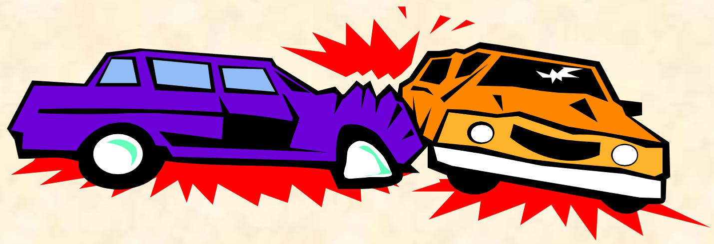
Collision Handling

- Collision = two keys hashing to same value
  - Collisions occur when different elements are mapped to the same cell
- Essentially unavoidable
  - you have a ridiculous amount of memory
- Challenge: efficiently cope with collisions

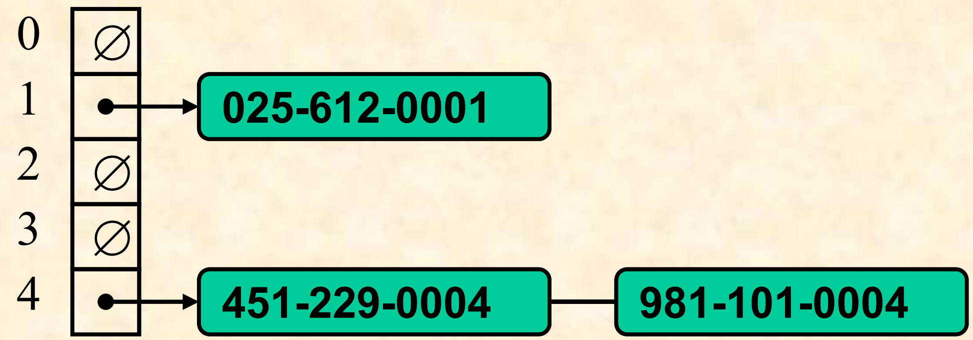
Collision Handing (contd)

- Collision Handing Techniques
  - Separate Chaining (array of M linked lists)
    - Hash: map key to integer i between 0 and M-1
    - Insert: put at front of ith chain (if not already there).
    - Search: only need to search ith chain.
    - Running time: proportional to length of chain.
  - Open Addressing
    - M much larger than N.
    - Plenty of empty table slots.
    - When a new key collides, find next empty slot and put it there..

Separate Chaining



- **Separate Chaining**: let each cell in the table point to a linked list of elements that map there
- M = number of linked lists
- N = number of keys
- M much smaller than N.
- N / M keys per table position.
- Put keys that collide in a list.
- Need to search lists.



- Chaining is simple, but requires additional memory outside the table

Separate Chaining Performance

- Search cost is proportional to length of chain.
- Trivial: average length = N / M.
- Worst case: all keys hash to same chain.

Linear Probing

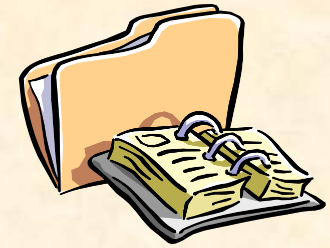
- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32 in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

		41			18	44	59	32	22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Search with Linear Probing



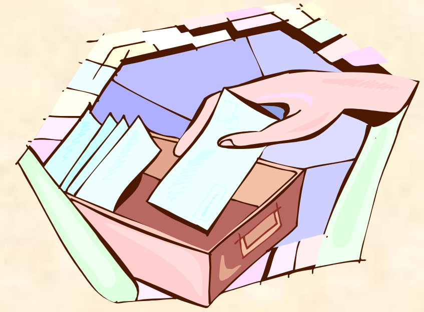
- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been probed unsuccessfully

Algorithm *findElement*(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *NO_SUCH_KEY*
        else if $c.key() = k$
            return *c.element*()
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return *NO_SUCH_KEY*

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called ***AVAILABLE***, which replaces deleted elements
- removeElement(***k***)
  - We search for an item with key ***k***
  - If such an item (***k, o***) is found, we replace it with the special item ***AVAILABLE*** and we return element ***o***
  - Else, we return ***NO_SUCH_KEY***
- insert Item(***k, o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until a cell ***i*** is found that is either
    - empty or
    - stores ***AVAILABLE***,
  - We store item (***k, o***) in cell ***i***

Double Hashing



- Suppose h maps some key k to a cell A[i] that is already occuppied { $i = h(k)$ }
- Double hashing uses a secondary hash function $\boldsymbol{d}(\boldsymbol{k})$ and handles collisions by placing an item in the first available cell of the series

$$(\boldsymbol{i} + \boldsymbol{j}\boldsymbol{d}(\boldsymbol{k})) \bmod N$$

  for $\boldsymbol{j} = 1, \ldots, N - 1$
- The secondary hash function $\boldsymbol{d}(\boldsymbol{k})$ cannot have zero values
- The table size $\boldsymbol{N}$ must be a prime to allow probing of all the cells
- Common choice of compression map for the secondary hash function:

$$\boldsymbol{d}_2(\boldsymbol{k}) = \boldsymbol{q} - \boldsymbol{k} \bmod \boldsymbol{q}$$

  where
  - $\boldsymbol{q} < N$
  - $\boldsymbol{q}$ is a prime
- The possible values for $\boldsymbol{d}_2(\boldsymbol{k})$ are

$$1, 2, \ldots, \boldsymbol{q}$$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
  - $(h(k) + jd(k)) \bmod 13$ for $j = 1, \ldots, 12$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

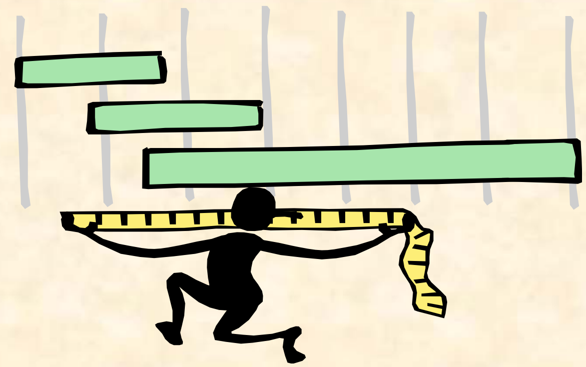
k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance of Open- Address Hashing



- In the worst case, searches, insertions and removals on a hash table take $\boldsymbol{O}(\boldsymbol{n})$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches