Fall 2017

Data Structures Lecture 4

Fang Yu

Software Security Lab.
Dept. Management Information Systems,
National Chengchi University

Abstract List Data Structures

Array Lists, Linked Lists, and Doubly Linked Lists

Array List

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

Array List

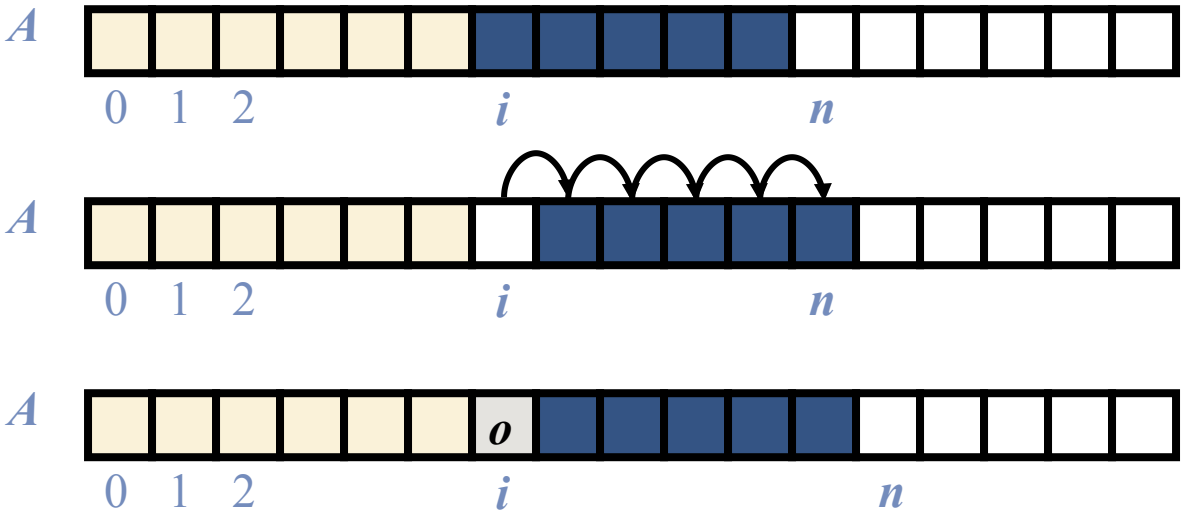
- Main methods:
  - get(integer i): returns the element at index i without removing it
  - set(integer i, object o): replace the element at index i with o and return the old element
  - add(integer i, object o): insert a new element o to have index i
  - remove(integer i): removes and returns the element at index i
- Additional methods:
  - size()
  - isEmpty()

Array-based Implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation $get(i)$ is implemented in $O(1)$ time by returning $A[i]$
- Operation $set(i,o)$ is implemented in $O(1)$ time by performing $t = A[i]$, $A[i] = o$, and returning t.

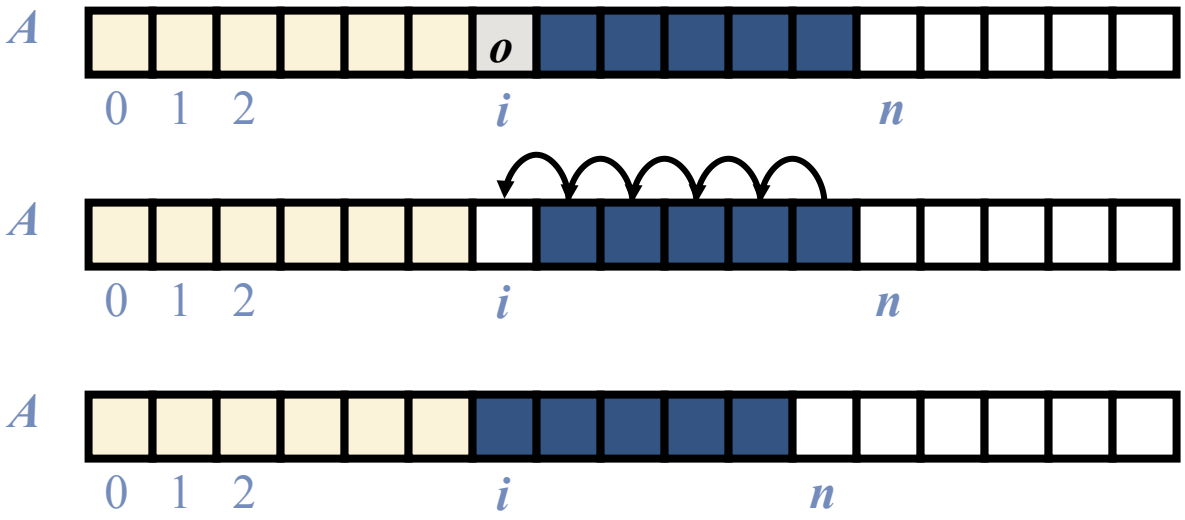
Insertion

- In operation ***add*(*i*, *o*)**, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

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



Performance

- In the array based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - *size*, *isEmpty*, *get* and *set* run in $O(1)$ time
  - *add* and *remove* run in $O(n)$ time in the worst case
- In an *add* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Growthable Array-based Array List

- In an add(o) operation (without an index), we always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Growthable Array-based Array List

```
Algorithm add(o)
    if t = S.length - 1 then
        A ← new array of
                size …
        for i ← 0 to n−1 do
            A[i] ← S[i]
        S ← A
    t ← t + 1
    S[t−1] ← o
```

//create a new array A (larger than S)

//copy the elements in S to A
//Replace S with A
//increase the size by 1
//add o as the last element

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

Doubling Strategy

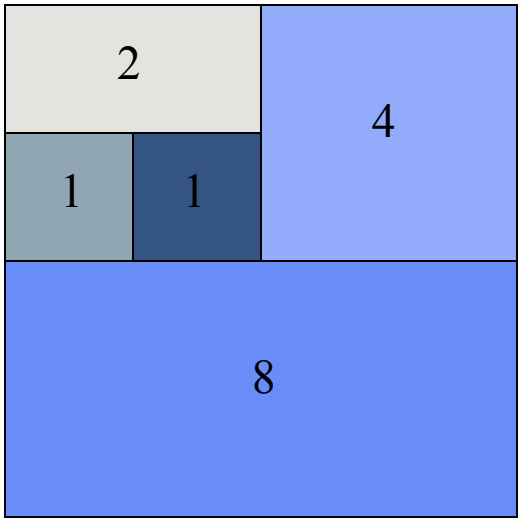
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

  $$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$

  $$3n - 1$$

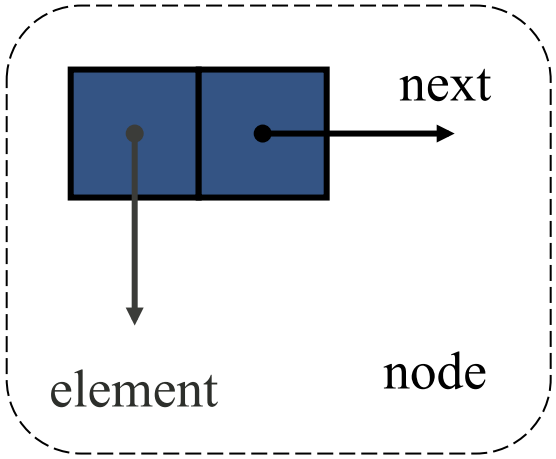
- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

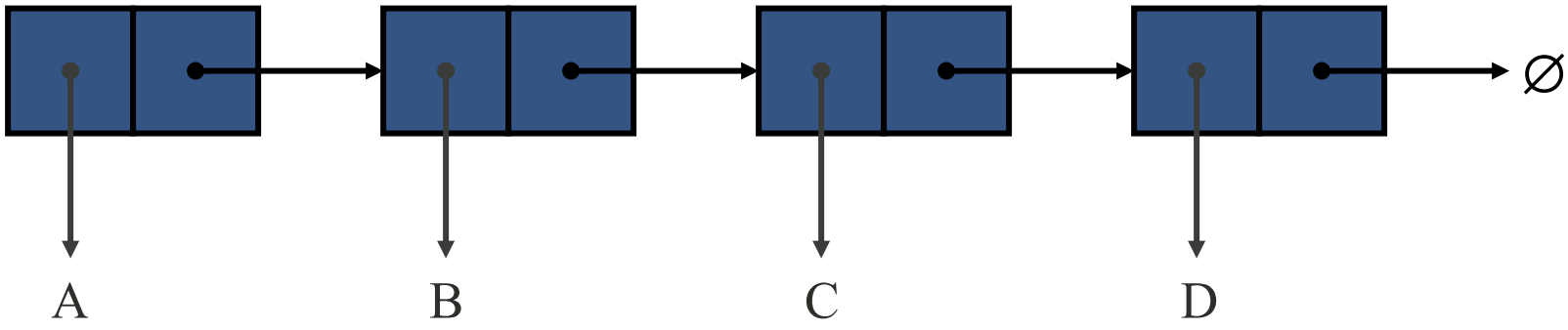
geometric series



Singly Linked List

- A concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



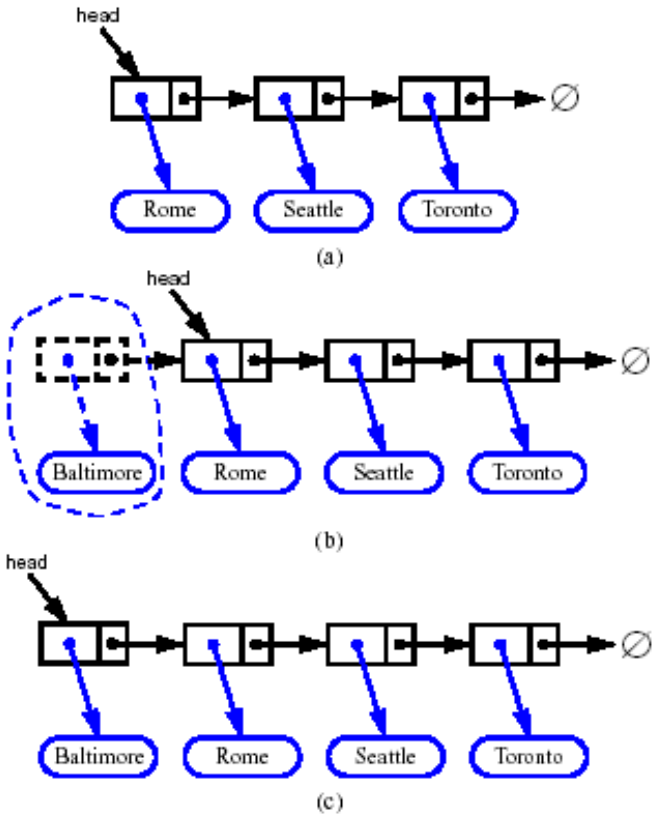


The Node Class for Singly Linked List

```
public class Node          {                    // Instance variables
  private  Object element;
  private  Node next;
  public  Node()           {                    // Creates a node with null
     this(null,  null);                         // references to its element and next node.
   }
  public  Node(Object e,  Node n)  {            // Creates a node with the given element
       element  =  e;                           // and next node.
       next  =  n;
  }
  public  Object getElement()  {                // Accessor methods
     return  element;
  }
  public  Node getNext()  {
     return  next;
  }
  public void  setElement(Object newElem)  {    // Modifier methods
       element  =  newElem;
  }
  public void  setNext(Node newNext)  {
       next  =  newNext;
  }
}
```

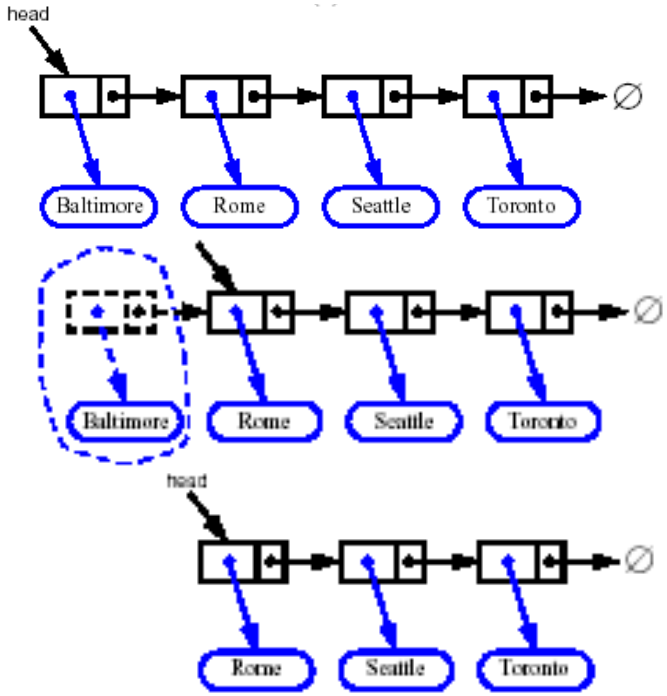
Inserting at the Head

1. Allocate a new node
2. Insert the new element
3. Have the new node to point to head
4. Update head to point to the new node



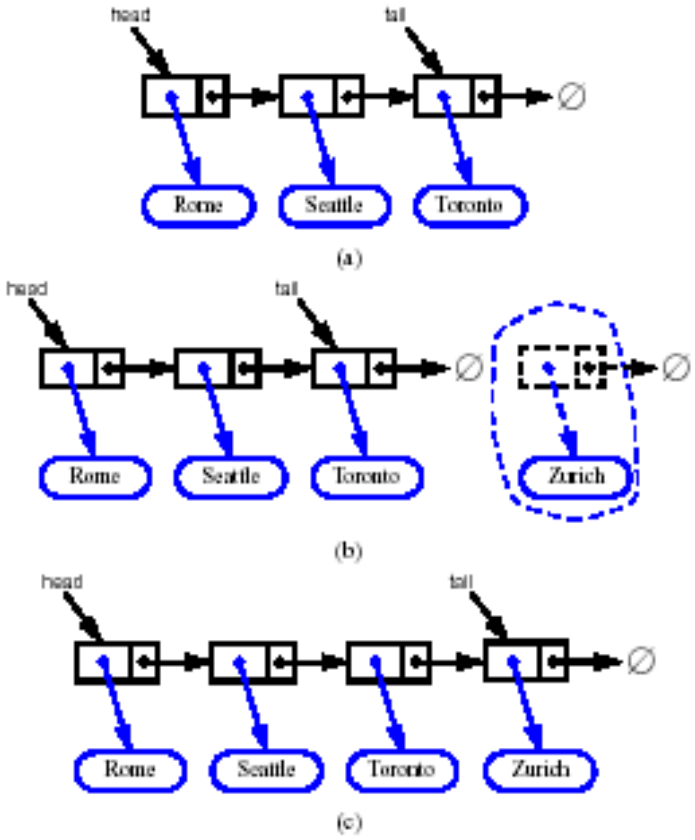
Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



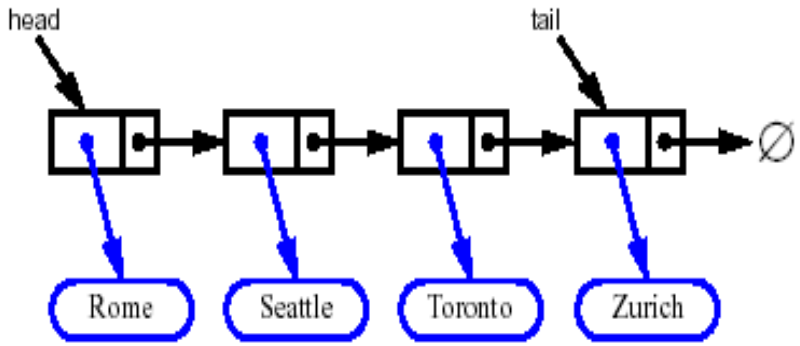
Inserting at the Tail

1. Allocate a new node
2. Insert the new element
3. Have the new node to point to null
4. Have the old last node to point to the new node
5. Update tail to point to the new node



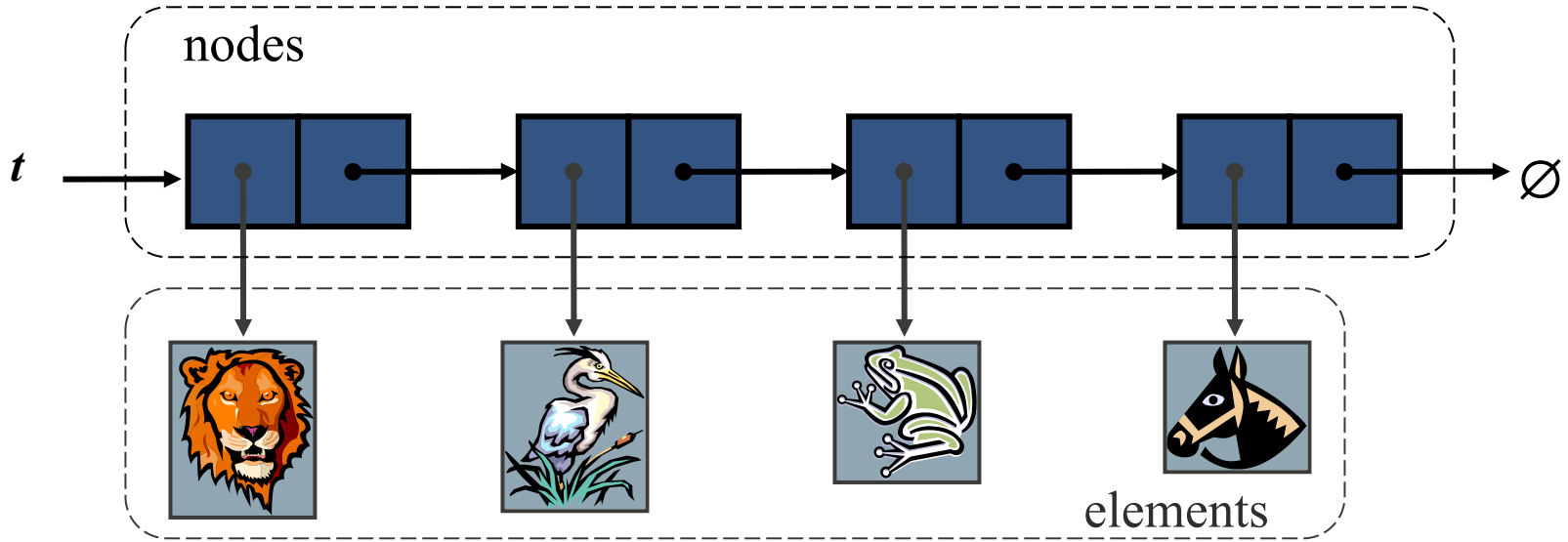
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



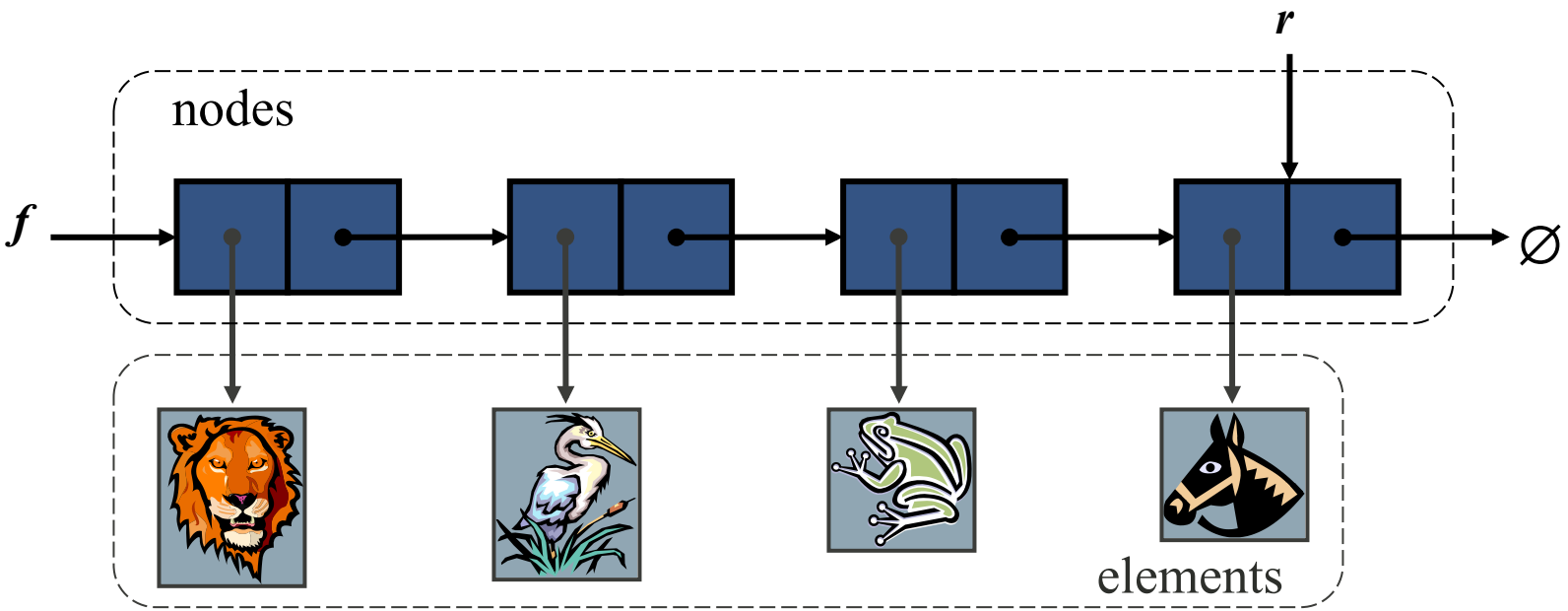
Stack as a Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Queue as a Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object element(): returns the element stored at the position

Node List ADT

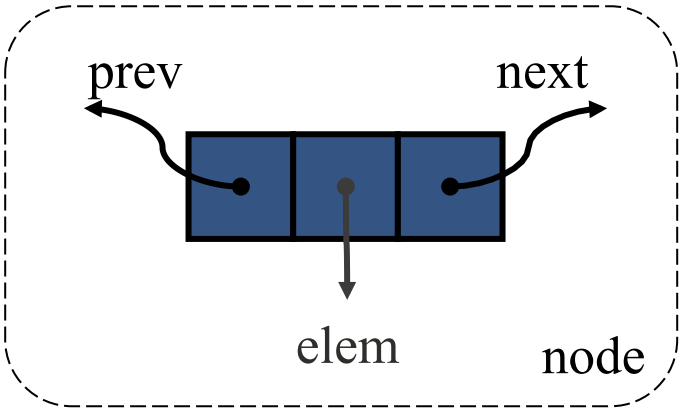
- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions

- Generic methods:
  - size(), isEmpty()
- Accessor methods:
  - first(), last()
  - prev(p), next(p)

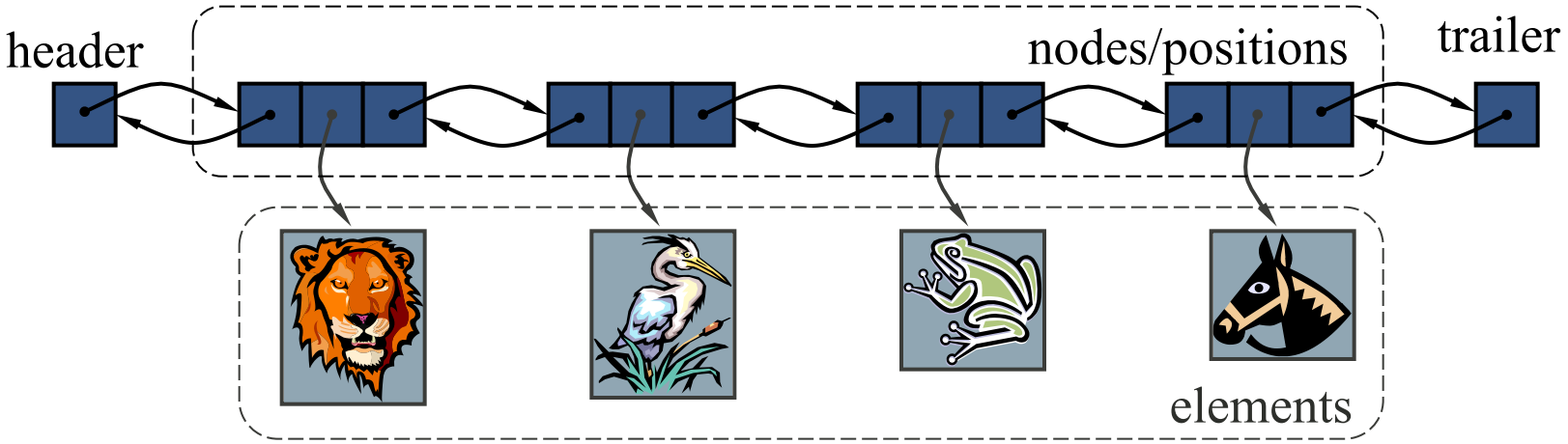
- Update methods:
  - set(p, e)
  - addBefore(p, e), addAfter(p, e),
  - addFirst(e), addLast(e)
  - remove(p)

Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

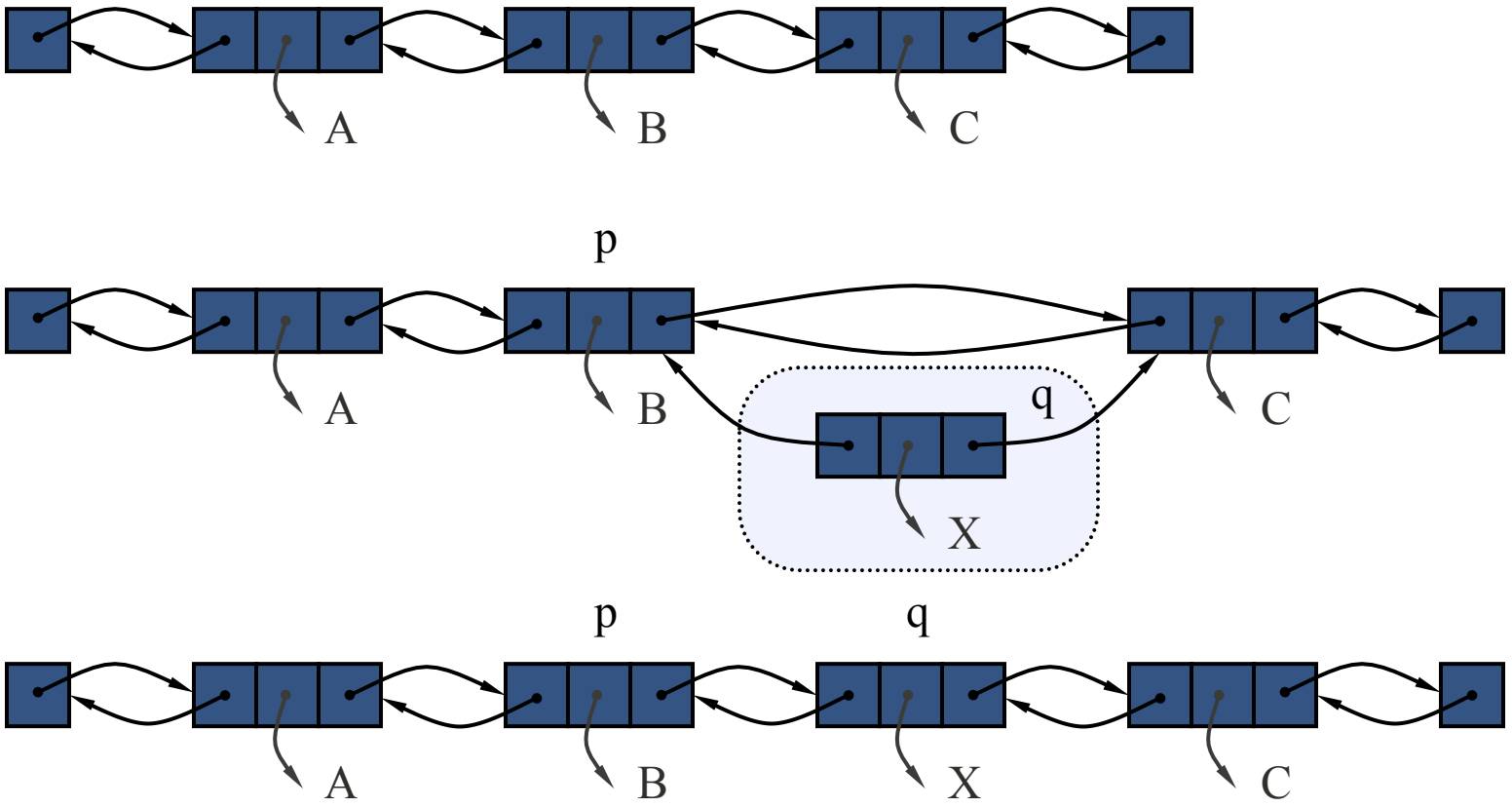


Doubly Linked List



Insertion

- We visualize operation insertAfter(p, X), which returns position q

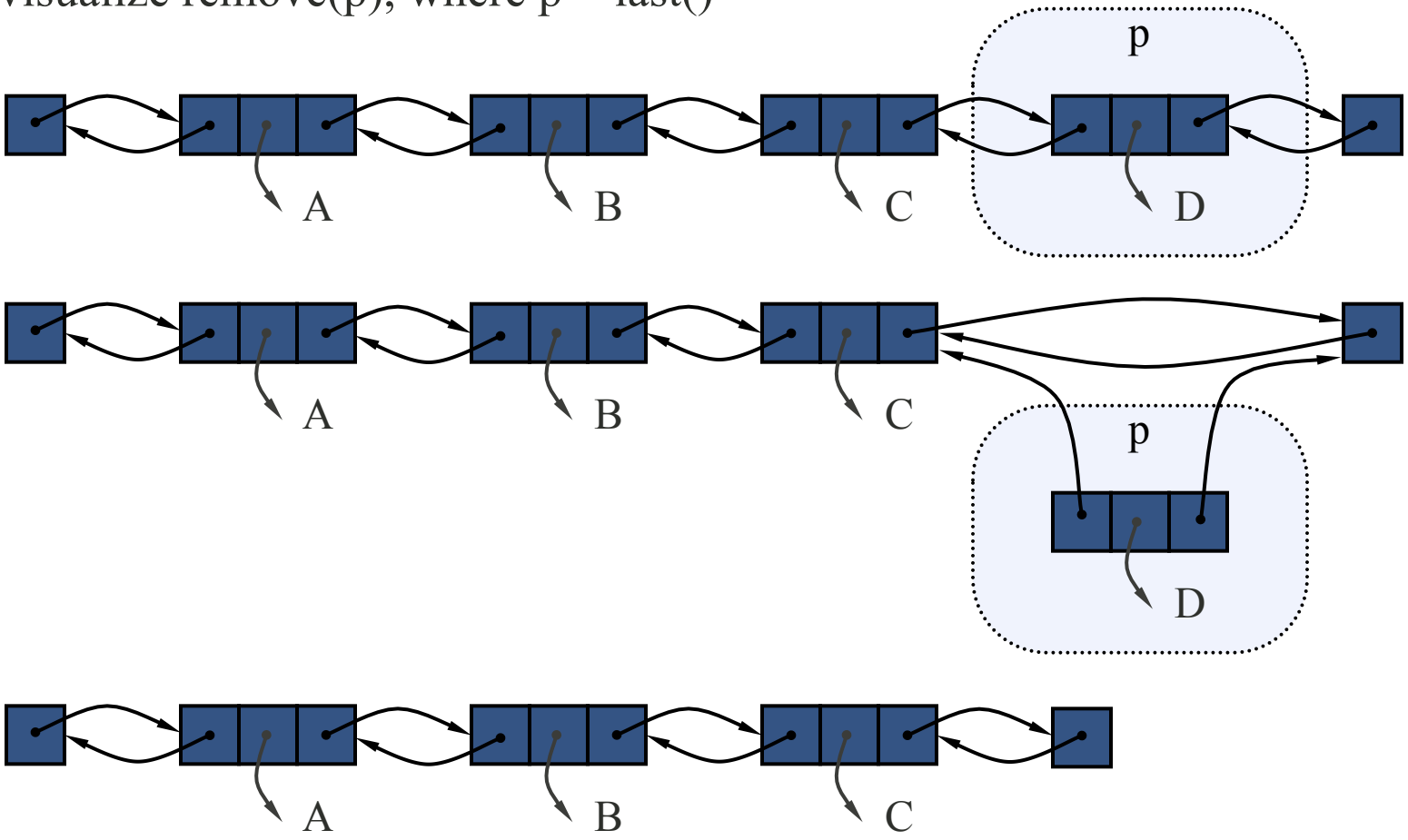


Insertion Algorithm

```
Algorithm addAfter(p,e):
    Create a new node v
    v.setElement(e)
    v.setPrev(p)                        //link v to its predecessor
    v.setNext(p.getNext())    //link v to its successor
    (p.getNext()).setPrev(v)  //link p's old successor to v
    p.setNext(v)                        //link p to its new successor, v
    return v                            //the position for the element e
```

Deletion

- We visualize remove(p), where p = last()



Deletion Algorithm

```
Algorithm remove(p):
    t = p.element          //a temporary variable to hold the return value
    (p.getPrev()).setNext(p.getNext()) //linking out p
    (p.getNext()).setPrev(p.getPrev())
    p.setPrev(null)                                  //invalidating the position p
    p.setNext(null)
    return t
```

HW4 (Due on 10/26)

Maintain an ordered keyword list.

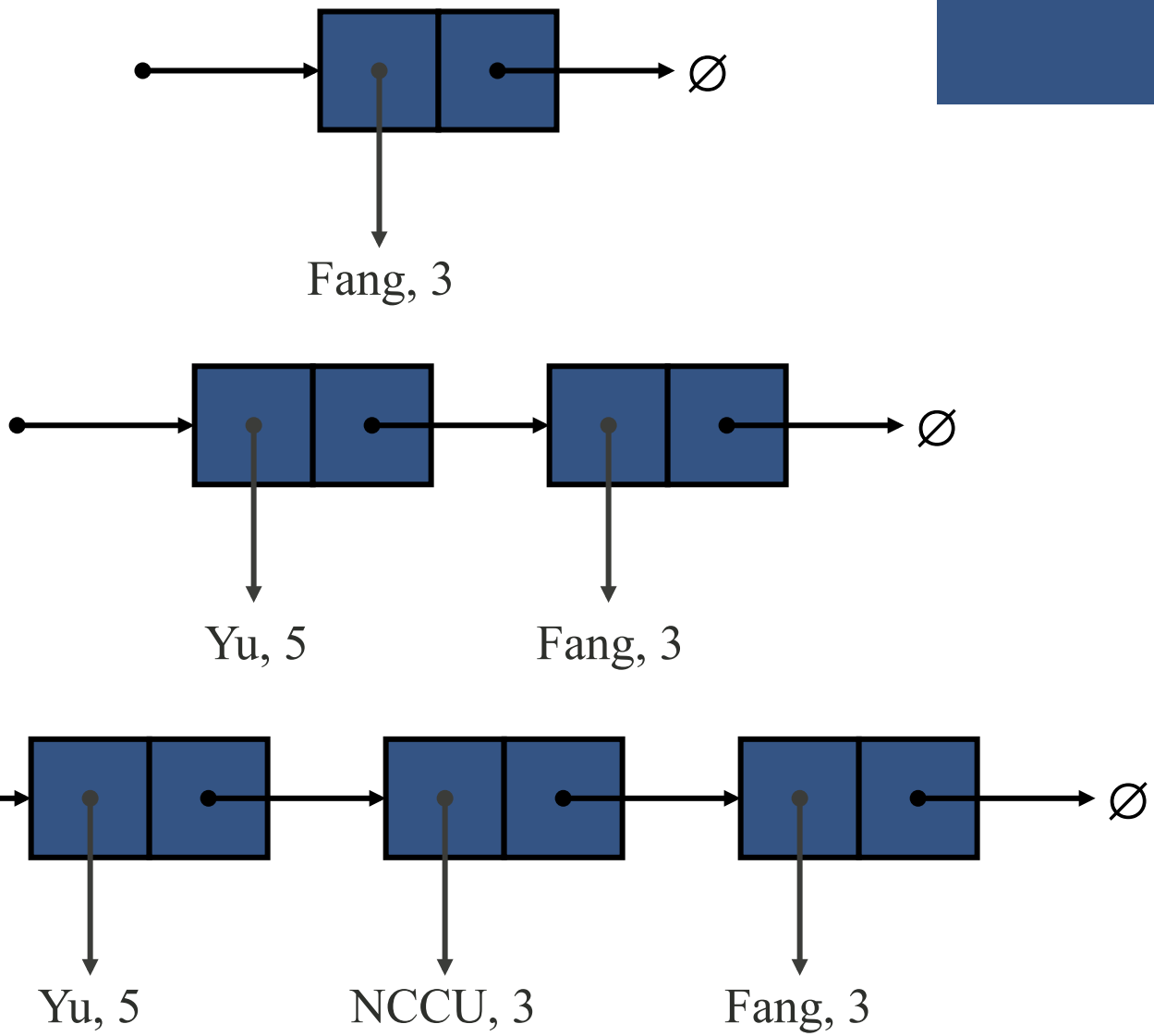
- A keyword is a tuple of [String name, Integer count, Double weight]
- Keep the list in order by its count while adding or deleting elements
- For the list structure, you can
  - Use java.util.ArrayList, or
  - java.util.LinkedList, or
  - Develop it by yourself
- Given a sequence of operations in a txt file, parse the txt file and execute each operation accordingly

Add and Output

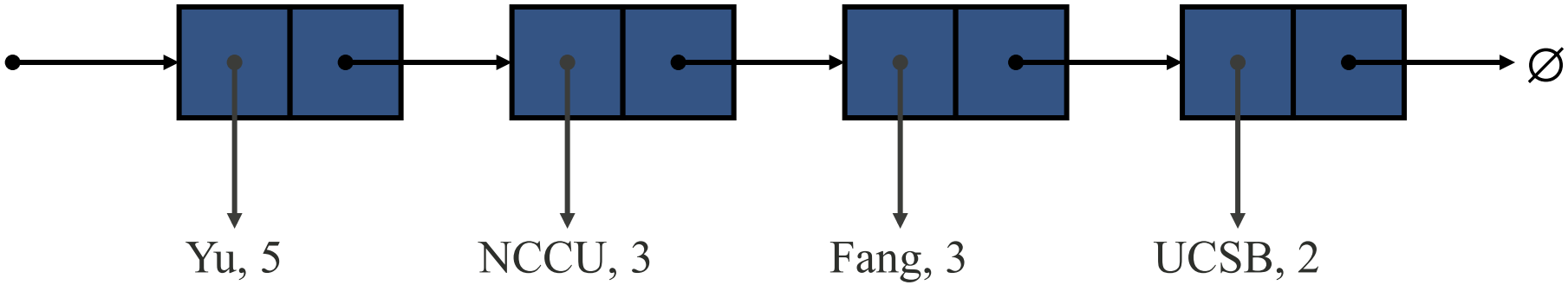
operations	description
add(Keyword k)	Insert k to the list in order
outputIndex(int i)	Output the ith keyword in the list
outputCount(int c)	Output all keywords whose count is equal to c
outputHas(string s)	Output all keywords whose name contains s
outputName(string s)	Output all keywords whose name is equal to s
outputFirstN(int n)	Output the first n keywords
outputScore()	Output the score of the whole list

Add in order

add Fang 3 0.5
add Yu 5 1.8
add NCCU 3 2.8
add UCSB 2 2.5
…



Output operations



…

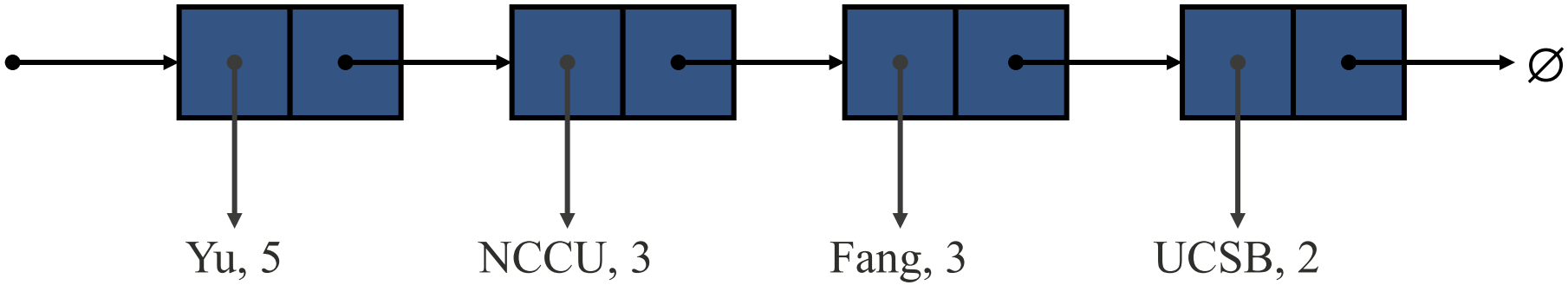
outputCount 3	[NCCU, 3] [Fang, 3]
outputName Yu	[Yu, 5]
outputHas U	[NCCU, 3] [UCSB, 2]
outputIndex 1	[Yu, 5]
outputFirstN 2	[Yu, 5] [NCCU,3]
outputScore	Score: 24.3

…

Delete

operations	description
deleteIndex(int i)	Delete the ith keyword in the list
deleteCount(int c)	Delete all keywords whose count is equal to c
deleteHas(string s)	Delete all keywords whose name contains s
deleteName(string s)	Delete all keywords whose name is equal to s
deleteFirst(int n)	Delete the first n keywords

Delete operations



deleteCount 3
deleteName Yu
deleteHas U
deleteIndex 1
deleteFirstN 2
deleteAll

An input file

1. You need to read the sequence of operations from a txt file
2. The format is firm
3. Raise an exception if the input does not match the format

```
add Fang 3 1.5
add Yu 5 1.2
add NCCU 3 0.8
add UCSB 2 2.2
add MIS 2 1.2
add Badminton 4 2.3
add Food 3 0.1
add Data 3 0.3
add Structure 4 2.1
outputScore
deleteCount 3
outputCount 2
outputName Yu
deleteName Yu
outputHas a
deleteHas a
outputIndex 2
deleteIndex 4
deleteFirstN 1
outputFirstN 3
deleteAll
```

Coming up…

- We will discuss stacks and queues on Oct. 26
- We will have programming tests in the lab on Nov. 6
- Read TB Chapter5 and Chapter 8