Asymptotic Notation in Practice

- The fastest algorithm in practice or for practical size input data sets is not always revealed!!!
- Because
  - Constants are dropped
  - Low-order terms are dropped
  - Algorithm efficiencies on small input sizes are not considered
- However, asymptotic notation is very effective
  - for comparing the scalability of different algorithms as input sizes become large

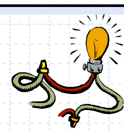
1

Relatives of Big-Oh

- **big-Omega**
  - f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≥ c•g(n) for $n \geq n_0$
  - f(n) is Ω(g(n)) if g(n) is an ***asymptotic lower bound*** on f(n)
- **big-Theta**
  - f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that c'•g(n) ≤ f(n) ≤ c''•g(n) for $n \geq n_0$
  - f(n) is Θ(g(n)) if g(n) is an ***asymptotic tight bound*** on f(n)
- These are sets of functions, also called classes of functions
- Exercise: express these complexity classes as sets of functions

2

Intuition for Asymptotic Notation



Big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

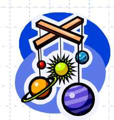
Big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

There are two other classes, Little-Oh and Little-Omega, but these are beyond the scope of this class and are rarely useful in algorithm analysis because we want a tight bound on running time (Theta).

3

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≥ c•g(n) for $n \geq n_0$

  let c = 5 and $n_0 = 1$
- **$5n^2$ is $\Omega(n)$**

  f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≥ c•g(n) for $n \geq n_0$

  let c = 1 and $n_0 = 1$
- **$5n^2 + 2$ is $\Theta(n^2)$**

  f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that c'•g(n) ≤ f(n) ≤ c''•g(n) for $n \geq n_0$

  Let c' = 5 and c'' = 7 and $n_0 = 1$, then c'•g(n) ≤ f(n) ≤ c''•g(n) for $n \geq n_0$

  That is, $5 \cdot n^2 \leq 5n^2 + 2 \leq 7 \cdot n^2$ for n ≥ 1 is true

4

Theta Limit Criteria

Suppose f(n) and g(n) are functions. Then

f(n) is Θ(g(n)) ("f(n) is theta of g(n)") if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = r$ for some nonzero number ***r***

[in this case, we can say, f(n) grows at the same rate as g(n)]

The classes of functions represented by Θ (also O and Ω) are called complexity classes.

Note: It is theoretically possible that limits of this kind may not exist. This situation almost never arises in the context of determining running times of algorithms, and so we do not attempt to handle this special case in this course.

5

Theta and Big-omega

- If f(n) *grows at least as fast* as g(n), we say f(n) is Ω(g(n)) ("big-omega")
  - That is, f(n) is Ω(g(n)) if g(n) is O(f(n)).
- If f(n) *grows at same rate* as g(n), we say f(n) is Θ(g(n)) ("theta")

6

Standard Complexity Classes

- The most common complexity classes used in analysis of algorithms are, in increasing order of growth rate:

  $\Theta(1), \Theta(\log n), \Theta(n^{1/k}), \Theta(n), \Theta(n\log n), \Theta(n^k)\ (k > 1),$

  $\Theta(2^n), \Theta(n!), \Theta(n^n)$

  Functions that belong to classes in the first row are known as *polynomial time bounded* because they are $O(n^k)$ for some $k \geq 0$.

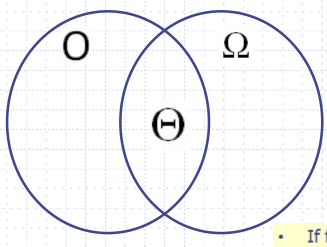
7

Examples

- Both $2n + 1$ and $3n^2$ are $O(n^2)$
- Both $2n^2 + 1$ and $3n^2$ are $\Theta(n^2)$
- Both $2n^2 – 1$ and $4n^3$ are $\Omega(n^2)$

8

Relationships Between the Complexity Classes



- If f(n) is in both O(g(n)) and Ω(g(n)), it is in Θ(g(n)).

9

Lecture 2: Stacks, Queues, Lists

Pure Knowledge Has
Infinite Organizing Power

10

Wholeness Statement

Knowledge of data structures allows us to pick the most appropriate data structure for any computer task, thereby maximizing efficiency.

Science of Consciousness: Pure knowledge has infinite organizing power, and administers the whole universe with minimum effort.

11

Review: Decomposition & Abstraction

- What do we mean by decomposition?
- What do we mean by abstraction?

12

Review:
Abstraction Mechanisms

- Abstraction by parameterization
- Abstraction by specification

13

Kinds of Abstractions

1. Procedural abstraction introduces new functions/operations
2. Data abstraction introduces new types of data objects (ADTs)
3. Iteration abstraction allows traversal of the elements in a collection without revealing the details of how the elements are obtained
4. Type hierarchy allows us to create families of related types
   - All members have data and operations in common that were defined in (inherited from) the supertype

14

What is a type?

15

Algorithms and Data Structures

- Closely linked
  - Algorithm (operation)
    - a step by step procedure for performing and completing some task in a finite amount of time
  - Data structure
    - an efficient way of organizing data for storage and access by an algorithm
- An ADT provides services to other algorithms
  - E.g., operations (algorithms) are embedded in the data structure (ADT)

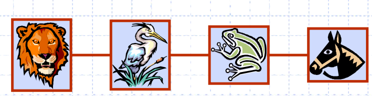
16

Abstract Data Types (ADTs)

- An ADT is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Today we are going to look at several examples:
  - Stack
  - Queue
  - List

17

List ADT
(with Singly Linked Nodes)



18

Outline and Reading

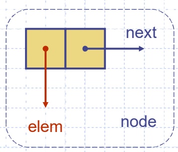
- Singly linked list

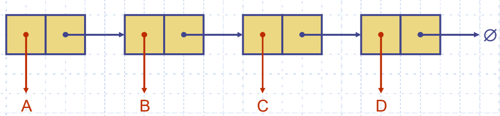
Linked List

- Motivation:
  - need to handle varying amounts of data
  - eliminate the need to resize the array
  - grows and shrinks exactly when necessary
  - efficient handling of insertion or removal from the middle of the data structure
  - random access is often unnecessary
- Built-in list data structures
  - Lisp, Scheme, ML, Haskell

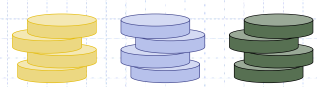
Singly Linked List

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





Stacks



Outline and Reading

- The Stack ADT
- Applications of Stacks

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out (LIFO) scheme
  - Like a spring-loaded plate dispenser
- Main stack operations:
  - void push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Exceptions

- Operations on the ADT may cause an error condition, called an exception
- Exceptions are said to be "thrown" when an operation cannot be executed
- Operations pop and top cannot be performed if the stack is empty
  - Attempting a pop or peek on an empty stack causes an EmptyStackException to be thrown

25

Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Evaluate an expression
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
- Linked List
  - Implementation is straightforward

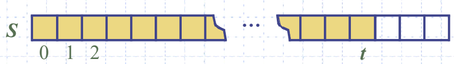
26

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- Elements are added from left to right
- A variable keeps track of the index of the top element

Algorithm *size*()
  return $t + 1$

Algorithm *pop*()
  if *isEmpty*() **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

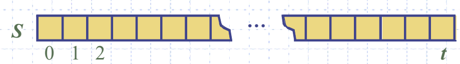


27

Array-based Stack (cont.)

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

Algorithm *push*(o)
  if $t = S.length - 1$ **then**
    throw *StackFullException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



28

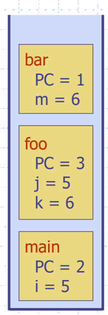
Runtime Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes onto the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  …
}
```



29

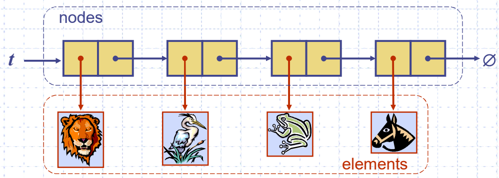
Performance and Limitations

- 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 at creation and cannot be changed
  - Trying to push a new element onto a full stack causes an implementation-specific exception

30

Stack with a Singly 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



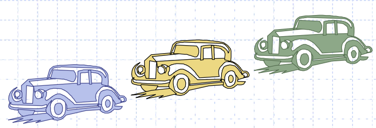
31

Main Point

1. Stacks are data structures that allow very specific and orderly insertion, access, and removal of their individual elements, i.e., only the top element can be inserted, accessed, or removed. *Science of Consciousness*: The infinite dynamism of the unified field is responsible for the orderly changes that occur continuously throughout creation.

32

Queues



33

Outline and Reading

- The Queue ADT
- Implementation with a circular array
- Queue interface in Java

34

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - void enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of remove or front on an empty queue throws an EmptyQueueException

35

Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming (OS)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

36

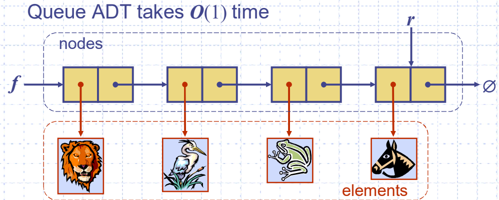
Main Point

2. The Queue ADT is a special ADT that supports orderly insertion, access, and removal. Queues achieve their efficiency and effectiveness by concentrating on a single point of insertion (end) and a single point of removal and access (front). *Science of Consciousness*: Similarly, nature is orderly, e.g., an apple seed when planted properly will yield only an apple tree.

- Are there any ADT's that could be implemented efficiently with a linked list?

Queue with a Singly Linked List

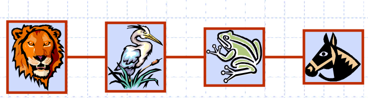
- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Queue ADT Implementation

- Can be based on either an array or a linked list
- Linked List
  - Implementation is straightforward
- Array
  - Need to maintain pointers to index of front and rear elements
  - Need to wrap around to the front after repeated insert and remove operations
  - May have to enlarge the array

List ADT (with Doubly Linked Nodes)



Outline and Reading

- Position ADT and List ADT
- Doubly linked list

Key Idea in Implementing a List

- Elements are accessed by Position
- Position is an ADT that models a particular place or location in a data structure
- We will use this abstraction in several data structures (today in the List ADT)
- Tomorrow in the Sequence ADT

43

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 or tree
- Just one method:
  - object element(): returns the element stored at the position

44

JavaScript Position ADT

```
class NPos {
    constructor (elem, prev, next) {
        // inserts this new node between prev and next
        this._elem = elem;
        this._prev = prev;
        this._next = next;
        if (prev != null) {
            prev._next = this;
        }
        if (next != null) {
            next._prev = this;
        }
    }
    element() {
        return this._elem;
    }
}
```


45

Fundamentals of Design

- **Fundamental Software Engineering Principles:**
  - Encapsulation
  - Loose Coupling
  - High Cohesion
  - Separation of Concerns
  - Subtyping & substitution
- **Fundamental Concepts:**
  - Instantiation
  - Abstraction
  - Implementation Hiding
  - Composition
  - Inheritance
  - Polymorphism

46

List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()
- Query methods:
  - isFirst(p), isLast(p)

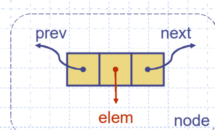
Accessor methods:
- first(), last()
- before(p), after(p)

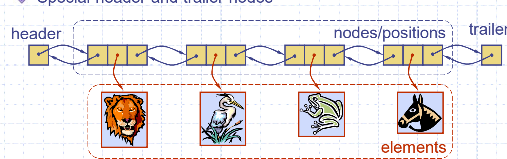
- Update methods:
  - replaceElement(p, e), swapElements(p, q)
  - insertBefore(p, e), insertAfter(p, e),
  - insertFirst(e), insertLast(e)
  - remove(p)

47

Doubly Linked List

- 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 header and trailer nodes

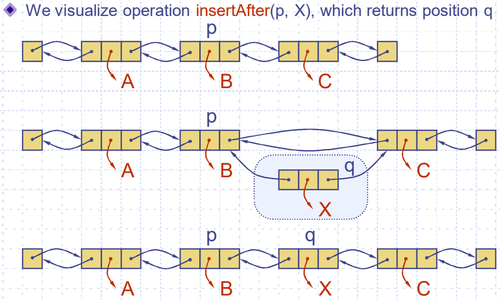




48

Insertion

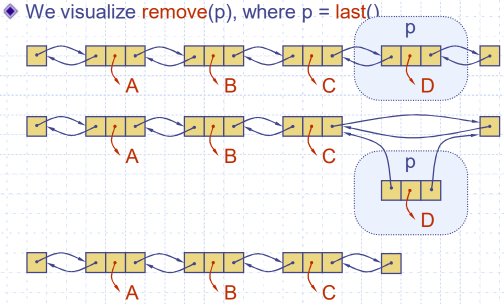
- We visualize operation insertAfter(p, X), which returns position q



49

Deletion

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



50

Performance of Linked List implementation of List ADT

- Generic methods:
  - size(), isEmpty()
- Query methods:
  - isFirst(p), isLast(p)
- Accessor methods:
  - first(), last()
  - before(p), after(p)
- Update methods:
  - replaceElement(p, e), swapElements(p, q)
  - insertBefore(p, e), insertAfter(p, e),
  - insertFirst(e), insertLast(e)
  - remove(p)

51

Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

52

Exercise on List

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
- Accessor methods:
  - position first()
  - position last()
  - position before(p)
  - position after(p)
- Query methods:
  - boolean isFirst(p)
  - boolean isLast(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
  - insertFirst(o)
  - insertLast(o)
  - insertBefore(p, o)
  - insertAfter(p , o)
  - remove(p)

Exercise:

- Write a method to calculate the sum of the integers in a list of integers
  - Only use the methods in the list to the left.

Algorithm sum(L)

53

Main Point

3. The algorithm designer needs to consider how a sequence of objects is going to be used because linked lists are much more efficient than arrays (vectors) when many insertions or deletions need to be made to random parts of a sequence (or list). *Science of Consciousness*: Nature always functions with maximum efficiency and minimum effort.

54

List Complexity

Operation		Positions
first(), last()		
before(p), after(p)		
replaceElement(p, o), swapElements(p, q)		
insertFirst(o), insertLast(o)		
insertAfter(p, o), insertBefore(p, o)		
remove(p)		

List Complexity

Operation		Positions
first(), last()		1
before(p), after(p)		1
replaceElement(p, o), swapElements(p, q)		1
insertFirst(o), insertLast(o)		1
insertAfter(p, o), insertBefore(p, o)		1
remove(p)		1

Exercise on List ADT

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
- Accessor methods:
  - position first()
  - position last(p)
  - position before(p)
  - position after(p)
- Query methods:
  - boolean isFirst(p)
  - boolean isLast(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
  - insertFirst(o)
  - insertLast(o)
  - insertBefore(p, o)
  - insertAfter(p , o)
  - remove(p)

Exercise:

- Given a List L, write a method to find the **Position** that occurs in the middle of L
  - Specifically, find middle as follows:
    - when the number of elements is odd return the position p in L such that the same number of nodes occur before and after p
    - when number of elements is even find p such that there is one more element that occurs before p than after
  - Do this without using a counter of any kind

Algorithm findMiddle(L)

Exercise on List ADT

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - objectIterator elements()
- Accessor methods:
  - position first()
  - position last(p)
  - position before(p)
  - position after(p)
- Query methods:
  - boolean isFirst(p)
  - boolean isLast(p)
- Update methods:
  - swapElements(p, q)
  - object replaceElement(p, o)
  - insertFirst(o)
  - insertLast(o)
  - insertBefore(p, o)
  - insertAfter(p , o)
  - remove(p)

Exercise:

- Given a List L, write a method to remove the element that occurs at the middle of L
  - Specifically, remove as follows:
    - when the number of elements is odd, remove the element e such that the same number of elements occur before and after e in L
    - when the number of elements is even, remove element e such that there is one more element that occurs after e than before
  - Return the element e that was removed; implement this without using a counter of any kind or any of the Random Access operations
  - Analyze the complexity of your solution

Algorithm removeMiddle(L)

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The List ADT may be used as an all-purpose class for storing collections of objects with only *sequential access* to its elements.
2. The underlying implementation of an ADT determines its efficiency depending on how that data structure is going to be used in practice.

3. **Transcendental Consciousness** is the unbounded, silent field of pure order and efficiency.
4. **Impulses within Transcendental Consciousness**: Within this field, the laws of nature continuously organize and govern all activities and processes in creation.
5. **Wholeness moving within itself**: In Unity Consciousness, when the home of all knowledge has become fully integrated in all phases of life, life is spontaneously lived in accord with natural law for maximum achievement with minimum effort.