Collections

Definition: A data structure that stores a collection of objects (elements)

The elements within a collection are usually **organized** based on:
-Order in which they were **added**
-Some **inherent** relationship

They can be linear or nonlinear

Needs a well defined **interface** to use properly

For each collection we examine, we will **consider**:
- How does the collection **operate** conceptually?
- How do we formally **define its interface**?
- What kinds of problems does it help us **solve**?
- What ways might we **implement** it?
- What are the **benefits and costs** of each implementation?

Operations that *define* how we **interact** with it:
They usually **include ways for** the user to:
-**add and remove** *elements, determine if the collection is* **empty**, *determine the collection's* **size**
They also may include:
-**iterators***, to process each element in the collection, operations that* **interact** with **other** *collections*

SET -> random selectoin, no orrder, no duplicates

STACK -> first in last out, adds to top, takes off top

QUEUE -> first in first out, adds to back, takes off frount

Collections (cont)

Rank and **Position** are 2 *different* ways to define the location of a particular element within the container

-For example, a list of **people** may be kept in **alphabetical** order by name or in the order in which they were **added** to the list
-Which type of collection you **use depends** on what you are trying to **accomplish**

Dynamic Memory and "new"

The operator **new dynamically** allocates memory from the **heap** (free memory) and returns a pointer

```
Candidate *c; //creates a pointer variable for Candidate structures
c = new Candidate; //actually allocates the memory for a Candidate data type
```

The new object will exist until it is explicitly de-allocated (no garbage collection!)
`delete Foo;`

Arrays can also be dynamically allocated in the same way, but must be de-allocated using the `delete[]`

If it has a `new` it needs a `delete`
It is essential to eventually de-allocate memory using delete that was allocated with new to avoid memory leaks, *once the pointer is gone you cant access it*

Analysis Tools

Write program and run it
clock it and plot it
Time X Input Size

We use the **Worst Case** not the Average Case
lo *Easier to analyze Crucial to applications such as games, finance and robotics*

Time is in unets were 1 is the time it would take for that RAM to acsess on pease of memory

By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size:

1.) count up primative opps, a loop from `i<-1 to n-1` is `2n`
2.) count each line up(adding them)`8n-3`
3.) then take the fastest growing part `8n`

--Growth Rate--

T(n) is afected by the hardwaer but the growth rate dose not chang, *growth rate is inhearet to the funtoin*

Growth rate is not afected by consatnts or lower odder terms

It's not usually **necessary to know the exact** growth function. The key issue is the **asymptotic complexity** *(how it grows as n increases)*. This is determined by the **dominant term** in the growth function
This is referred to as the **order** of the algorithm. We often use **Big-Oh** notation to specify the order

--Asymptotic Algorithm Analysis--

The asymptotic analysis of an algorithm determines the **running time** *in big-Oh notation*

Analysis Tools (cont)

The asymptotic analysis:
1.) We find the worst-case number of primitive operations executed as a function n(input size)
2.) We express this function with big-Oh notation

--Big-Oh--

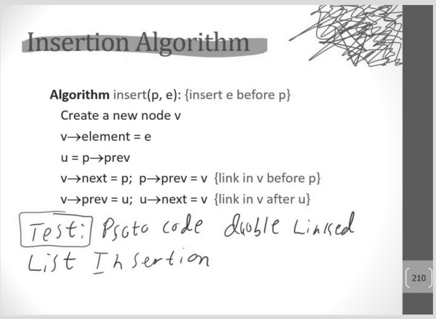
If is f(n) is of degree d, then f(n) is **O(n^d)**
-Use the smallest possible class of functions
-Use the simplest expression of the class

~**Loops**~
-A **loop executes** a certain number of times: n
-It contains the inner complexity of: m
Then the loop's **complexity** is n*m
If m is a **constant** -> O(n)
If m is a **function of n**(like another loop(n, n-1 or n/2)) -> O(n*m)*(simplified)*

~**Recursive**~
-The **size** of the problem is: n
-*Except for the base case*, each **recursive call** results in calling itself m more: m-1
So the **complexity** is m^n-1 or O(m^n)

-We pretend the memory is unlimited
-(Big-Oh)Since constant factors and lower--order terms are eventually dropped we can skip counting primitive operations

Double Linked List Insertoin Algorithom



Terms

data type	the programming constructs used to implement a collection
abstract data type	a data type whose values and operations are not inherently defined in a programming language
data structure	a group of values and the operations defined on those values
Algorithm	a step-by-step procedure for preforming some task in a finite amount of time

Abstraction

An abstraction hides certain details at certain times

It provides a way to deal with the complexity of a large system

A collection, like any well-defined object, is an abstraction

We want to separate the interface of the collection (how we interact with it) from the underlying details of how we choose to implement it

Data Types

Enumerations	User defined types for discrete values (behave much like integers) Default, numbered 0, 1, etc, but can specify values *enum Day { WINTER, SPRING, SUMMER, FALL } ;* *enum Day { FALL = 3, WINTER = 2, SUMMER = 1, SPRING = 4 } ;*

Abstract Data Types (ADTs)

Is an abstraction of a data structure

An ADT **specifies**:
-**Data** stored
- **Operations** on the data
- **Error conditions** associated with operations

No specification of how, just a list of operations. We should **hide the implementation** . . . The user of the ADT does **not** need to know the **details**, **just** how to **use** it. *Implementations may change* due to hardware or system upgrades*user doesn't need to see that*

The **container** (the data structure), and how that container is **manipulated**, is in many ways **more important** than the actual **data**. **Templates** allow C++ programs to manipulate **many different types** of data using the **same semantics**.

-Templates- allow C++ programs to manipulate **many different types** of data using the **same semantics**.

Abstract Data Types (ADTs) (cont)

Example: ADT modeling a simple stock trading system:
-The data **stored** are buy/sell orders
-The **operations** supported are
order **buy**(stock, shares, price)
order **sell**(stock, shares, price)
void **cancel**(order)
-**Error** conditions:
Buy/sell a **nonexistent** stock
Cancel a **nonexistent** order

template<typename E>

POINTERS

* - dereferencing (accesses the objects value **from** its **address**)

& - **address of** (returns the address of an object in memory)

Example: if int x, then &x will return the address of the x variable
Example: if int q, then q = &x and you can use *q = 5 effectively changes the value of x.*

```
int a = {12,15,18}; //init-
ializes the array a with size 3,
index positions 0-2, and
//values 12, 15 and 18
Int* p = a; //p points to a[0]
Int* q = &c[0]; //q also points
to a[0]
```

pointer and arrays

int *r[17]; creates an array of 17 int pointer elements
Once the array has been initialized, you can dereference any particular pointer
r[6] will dereference the 7th pointer in the array

Rank

Is **defined** as the **location** of an element within its container

first rank is 1 *not 0*

The index is typically one less than the rank.

The **index** value typically indicates how many elements precede that particular element
the **Rank** shows what spont it is in

Used in **Vectors***(it's really like indext it just shows what it is at not how manny more there are)*

Position

The concept of Position models the notion of **place within a data structure** where a single object is stored
Does not rely on the idea of rank

The Position ADT has one **method**:
Object **p.element**(): returns the element at **position** p
In C++ it is convenient to implement this as ***p**

Like nabors consers **what is around** not were it is

Useed in **Nodes** *(shows what it is colsed to, but not nesarly were it is)*

OVERALL VIEW

STL Container	*Description*
vector	Vector
deque	Double ended queue
list	List
stack	Last-in, first-out stack
queue	First-in, first-out queue
priority_queue	Priority queue
set (and multiset)	Set (and multiset)
map (and multimap)	Map (and multi-key map)

Stack ADT

The Stack ADT stores arbitrary objects

Insertions and deletions follow the **last-in first-out** scheme
Think of a **spring-loaded dispenser**

--Main stack operations--:
push(object): inserts an element
object **pop**(): removes 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 **empty**(): indicates whether no elements are stored

pop -> -
push -> +

C++ interface corresponding to our Stack ADT Uses an exception class StackEmpty Different from the built-in C++ STL class stack

-Direct applications:

Page-visited **history** in a Web browser

Undo sequence in a text editor

Chain of method calls in the **C++ run-time system**

-Indirect applications:

*Auxiliary data structure for algorithms

Component of other data structures*

Queue ADT

Stores arbitrary objects

Insertions and deletions follow the first-in first-out scheme
Insertions are at the rear of the queue and removals are at the front of the queue

-Main queue operations-
 enqueue(object): inserts an element at the end of the queue
 Dequeue(): removes 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 **empty**(): indicates whether no elements are stored

-Exceptions-
 Attempting the execution of dequeue or front on an empty queue throws an QueueEmpty

enqueue -> +
dequeue -> -
head -> retuns top(dose not chang anything)

C++ interface corresponding to our Queue ADT Requires the def-inition of exception QueueEmpty No corresponding built-in C++ class

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

Deque ADT

stores arbitrary objects

Insertions and deletions can be done to the front OR the back of the deque

-Main queue operations-
 insertFront(object): inserts an element at the front of the deque
 insertBack(object): inserts an element at the back of the deque
 eraseFront(): removes the first element of the deque
 eraseBack(): removes the last element of the deque

-Auxiliary deque operations-
 object **front**(): returns the element at the front without removing it
 object **back**(): returns the element at the back without removing it
 integer **size**(): returns the number of elements stored
 boolean **empty**(): indicates whether no elements are stored

-Exceptions-
 Attempting the execution of eraseFront, eraseBack, front or back on an empty deque throws an DequeEmptyException

insertFront -> +
insertBack -> +
eraseFront -> -
eraseBack -> -
front **->** retuns the frount element(dose not chang anything)
back **->** retuns the back element(dose not chang anything)

can be used as a stack and as a queue

Array List(Vector)

The **Vector or Array List** ADT extends the notion of array by **storing a sequence of objects**

--**Main methods**--
At(integer i): returns the element **at index i without removing** it
Set(integer i, object o): **replace** the element at index i with o
Insert(integer i, object o): **insert** a new element o to have index i
Erase(integer i): **removes** element at index i

--**Additional methods**--
Size()
Empty()

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)

A **major weakness** in array implementations of collections is the **fixed capacity** N for the number of elements that may be stored in the array.
Thus we double the array size when the array is full

Iterators

extends the concept of position by adding a traversal capability

An iterator behaves like a pointer to an element
***p -> returns the element** referenced by this *iterator*
++p -> advances to the *next element*
--p -> regresses to the *previous element*

Node List

The Node List ADT models a **sequence of positions** storing arbitrary objects

--**Generic methods**--
size(),
empty()

--**Iterators**--
begin(), end()

--**Update methods**--
insertFront(e),
insertBack(e)
removeFront(),
removeBack()

--**Iterator-based update**--
insert(p, e)
remove(p)

It establishes a **before/after relation** between *positions*

Sequences

The Sequence ADT is the **union** of the **Array** List and **Node** List ADTs

--**Generic methods**-
size(),
empty()

--**ArrayList-based methods**--
at(i),
set(i, o),
insert(i, o), erase(i)

--**List-based methods**--
begin(),
end()
insertFront(o),
insertBack(o)
eraseFront(),
eraseBack()
insert (p, o),
erase(p)

--**Bridge methods**-
atIndex(i),
indexOf(p)

The Sequence ADT is a basic, **general-purpose, data structure** for storing an **ordered** collection of elements