CSCE 310J: Data Structures & Algorithms

Abstract Data Types and Basic Data Structures

Dr. Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE310J

1

CSCE 310J: Data Structures & Algorithms

- **Giving credit where credit is due:**
  - **Most of slides for this lecture are based on slides created by Dr. Ben Choi, Louisiana Technical University.**
  - **I have modified them and added new slides**

2

Abstract Data Type

- **Abstract Data Type**
  - ***Data Structure*** **declaration**
  - ***Operations*** **performed on the data structure**
    - **e.g., create, destroy, or manipulate**
    - **These are logical operations that are independent of the actual implementation!**
- **Provides *data encapsulation* (information hiding)**
- **An ADT is implemented as a Class in languages such as C++ and Java**
- **Algorithms can be designed, specified, and proven correct using the logical properties of ADTs**
- **Performance analysis depends on the implementation!**

3

ADT Specification

- **The specification of an ADT describes how the operations (functions, procedures, or methods) behave in terms of Inputs and Outputs**
- **A specification of an operation consists of:**
  - **Calling prototype**
  - **Preconditions**
  - **Postconditions**
- **The calling prototype includes**
  - **name of the operation**
  - **parameters and their types**
  - **return value and its types**
- **The preconditions are statements**
  - **assumed to be true when the operation is called.**
- **The postconditions are statements**
  - **assumed to be true when the operation returns.**

4

Operations for ADT

- **Constructors**
  - **create a new object and return a reference to it**
- **Access functions**
  - **return information about an object, but do not modify it**
- **Manipulation procedures**
  - **modify an object, but do not return information**
- ***Destructors***
  - **deallocate an object**

5

More on ADTs

- State **of an object**
  - **current value of its data**
- **Some books claim that constructors and manipulation procedures should be described in terms of Access functions**
  - **Maybe…**
- **Recursive ADT**
  - **if any of its access functions returns the same class as the ADT**

6

ADT Design for Lists

IntList nil //constant denoting the empty list.

IntList constructList(int newElement, IntList oldList)
Precondition: None.
Postconditions: If newList = constructList(newElement, oldList) then
1. newList refers to a newly created list object;
2. newList ≠ nil;
3. first(newList) = newElement;
4. rest(newList) = oldList

int first(IntList aList) // access function
Precondition: aList ≠ nil
Postcondition: if element = first(aList) then
1. element ≠ nil

IntList rest(IntList aList) // access fcn
Precondition: aList ≠ nil

7

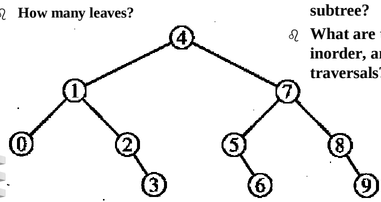
Binary Tree

- **A binary tree T is a set of elements, called nodes, that is empty or satisfies:**
  1. There is a distinguished node *r* called the root
  2. The remaining nodes are divided into two disjoint subsets, L and R, each of which is a binary tree.
     – L is called the left subtree of T and R is called the right subtree of T.
- **There are at most 2^d nodes at depth *d* of a binary tree.**
- **A binary tree with *n* nodes has height at least Ceiling[lg(n+1)] – 1.**
- **A binary tree with height *h* has at most 2^{h+1} –1 nodes**

8

Binary Tree Example

- **What node is the root?**
- **What is the depth of each node?**
  - How many nodes are there at that depth?
  - Is this the maximum number of nodes at that depth?
- **How many internal nodes?**
- **How many leaves?**



- **What is the value of *n*?**
- **What is the height of the tree?**
- **What is the minimum (maximum) height of a binary tree with this many nodes?**
- **What is the height of each subtree?**
- **What are the preorder, inorder, and postorder traversals?**

9

Stacks

- **A stack is a linear structure in which insertions and deletions are always made at one end, called the top.**
- **This updating policy is called last in, first out (LIFO).**
- **Operations:**
  - Stack create()
  - boolean isEmpty(Stack *s*)
  - Object top(Stack *s*),
  - void push(Stack *s*, Object *e*),
  - void pop(Stack s)

10

Queue

- **A queue is a linear structure in which**
  - all insertions are done at one end, called the rear or back, and
  - all deletions are done at the other end, called the front.
- **This updating policy is called first in, first out (FIFO).**

11

Priority Queue

- **A priority queue is similar to a FIFO queue but different…**
  - element order is related to each element's priority, rather than its chronological arrival time.
- **"As each element is inserted into a priority queue, *conceptually* it is inserted *in order of* its priority."**
- **The one element that can be inspected and removed is the most *important element* currently in the priority queue.**
  - a cost viewpoint: the smallest priority
  - a profit viewpoint: the largest priority
- **Priority queue operations are not in Θ(1). Their complexity varies depending on the implementation, as we shall see.**

12

Set: first some basics...

- A set is a collection of distinct elements.
- The elements are of the same "type"
- "element e is a member of set S" is denoted as $e \in S$
- Read "e is in S"
- A particular set is defined by listing or describing its elements between a pair of curly braces: $S_1 = \{a, b, c\}$, $S_2 = \{x \mid x \text{ is an integer power of } 2\}$ read "*the set of* all elements x *such that* x is …"
- $S_3 = \{\} = \emptyset$, has no elements, called empty set
- A set has no inherent order.

13

Subset, Superset; Intersection, Union

- If all elements of one set, S_1, are also in another set, S_2,
- Then S_1 is said to be a *subset* of S_2, $S_1 \subseteq S_2$ and S_2 is said to be a *superset* of S_1, $S_2 \supseteq S_1$.
- Empty set is a subset of every set, $\emptyset \subseteq S$
- *Intersection*

  $S \cap T = \{x \mid x \in S \text{ and } x \in T\}$
- *Union*

  $S \cup T = \{x \mid x \in S \text{ or } x \in T\}$

14

Sequence

- A group of elements in a *specified order* is called a sequence.
- A sequence can have repeated elements.
- Sequences are defined by listing or describing their elements in order, enclosed in parentheses.
  - e.g. S1 = (a, b, c), S2 = (b, c, a), S3 = (a, a, b, c)
- A sequence is *finite* if there is an integer *n* such that the elements of the sequence can be placed in a one-to-one correspondence with (1, 2, 3, …, *n*).
- If all the elements of a finite sequence are distinct, that sequence is said to be a *permutation* of the finite set consisting of the same elements.
- One set of *n* elements has *n*! distinct permutations.

15

Cardinality

- A set, S, is *finite* if there is an integer *n* such that the elements of S can be placed in a one-to-one correspondence with {1, 2, 3, …, *n*}
  - in this case we write $|S| = n$
- How many distinct subsets does a finite set on *n* elements have?
  - There are 2^n subsets.
- How many distinct subsets of cardinality k does a finite set of n elements have?
  - There are C(n, k) = n!/((n-k)!k!), "n choose k"

$$\binom{n}{k}$$

16

Tuples and Cross Product

- A tuple is a finite sequence.
  - Ordered pair (x, y), triple (x, y, z), quadruple, and quintuple
  - A *k*-tuple is a tuple of *k* elements.
- The *cross product* of two sets, say S and T, is $S \times T = \{(x, y) \mid x \in S, y \in T\}$
- $|S \times T| = |S|\,|T|$
- It often happens that S and T are the same set,
  - e.g. $N \times N$ where N denotes the set of natural numbers, {0,1,2,…}

17

Relations and Functions

- A *relation* is some subset of a (possibly iterated) cross product.
- A binary relation is some subset of a simple cross product, e.g. $R \subseteq S \times T$
  - The "less than" relation can be defined as $\{(x, y) \mid x \in N, y \in N, x < y\}$
- Important properties of relations; let $R \subseteq S \times S$
  - reflexive: for all $x \in S$, $(x, x) \in R$.
  - symmetric: if $(x, y) \in R$, then $(y, x) \in R$.
  - antisymmetric: if $(x, y) \in R$, then $(y, x) \notin R$
  - transitive: if $(x,y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.
- A relation that is reflexive, symmetric, and transitive is called an *equivalence relation*,
  - I.e., partitions the underlying set S into equivalence classes $S_1, S_2, \ldots$ s.t. elements with in S_i are equivalent to each other.
- A *function* is a relation in which no element of S (of S x T) is repeated with the relation. (informal def.)

18

Union-Find ADT for Disjoint Sets

- **Through a *Union* operation, two (disjoint) sets can be combined.**
  - **(to insure the disjoint property of all existing sets, the original two sets are removed and the new set is added)**
  - **Let the set ids of the original two sets be, *s* and *t*, $s \neq t$**
  - **Then, the new set has a unique set id that is neither *s* nor *t*.**
- **Through a *Find* operation, the current *set id* of an element can be retrieved.**
- **Often elements are integers and the set id is some particular element in the set, called the leader, as in the example in the book**

19

Union-Find ADT Example

Notice how this ADT differs from the book!

- **unionFind create(int n)**
  - **// create a set of *n* singleton disjoint sets {{1},{2},{3},…,{n}}**
- **setId find(UnionFind sets, int element)**
  - **// return the set id for *element***
- **void makeSet(unionFind sets, int element)**
  - **//union one singleton set {e} (e not already in the sets)**
    // into exiting sets
- **void union(unionFind sets, setId s, setId t)**
  - **// s ≠ t**
  - **// a new set is created by union of set [s] and set [t]**
  - **// the new set id is either s or t, in some cases min(s, t)**

20

Dictionary ADT

- **A dictionary is a general associative storage structure.**
- **Items in a dictionary**
  - **have an identifier, and**
  - **associated information that needs to be stored and retrieved.**
- **No order is implied for identifiers in a dictionary ADT**
- **The Dictionary ADT is useful in dynamic programming, which is covered later in the semester.**

21