Closed Lists
and other
Data Structures

Robert Keller
October 2012
Open vs. Closed Lists

Two general list models:

- Open lists:
  - Elements and sublists can be shared
  - Mutation of lists is discouraged
  - Mathematically elegant

- Closed lists:
  - Sharing generally not done
  - Mutation of lists is ok, because they are encapsulated
  - Mathematically less attractive

Closed lists can be built by wrapping open lists
Abstract Data Types (ADT’s)

“Abstract Data Type” means:
- A data domain, with
- Operations on elements of the domain

This terminology has faded a bit with the increase in object-oriented programming, but the viewpoint is still worthwhile.

Both Open and Closed Lists can be used to implement various Abstract Data Types
Example: Stack ADT

- **Domain**: Arbitrary
- **Main Operations**:
  - `void push(Object)`
  - `Object pop()`
  - `boolean isEmpty()`
Example: Queue ADT

- **Domain:** Arbitrary
- **Main Operations:**
  - `void enqueue(Object)`
  - `Object dequeue()`
  - `boolean isEmpty()`
Implementing Stacks and Queues

- Try using an OpenList
Application of Stacks and Queues

- Stacks are commonly used “under the hood” to implement recursion.
  - The arguments and other variables for successive recursive calls are stacked, so that they can be used following the return from the inner call.

- Stacks and Queues are useful in two distinct varieties of tree or graph search.
Depth- vs. Breadth- First in Graph

Example: Find even-numbered nodes

1  2  3  4  5  6  7  8

Diagram showing the graph with even-numbered nodes.
Depth- vs. Breadth- First in Graph

Example: Find even-numbered nodes

Depth-First

Breadth-First

start 1 start 1

2 8

3 5 7

4

2 8

3 5 7

4
Depth- vs. Breadth-First in Graph

Example: Find even-numbered nodes

Depth-First: 2

Breadth-First: 2
Depth- vs. Breadth- First in Graph

Example: Find even-numbered nodes

Depth-First: 2

Breadth-First: 2 8
Depth- vs. Breadth- First in Graph

Example: Find even-numbered nodes

Depth-First: 2 4

Breadth-First: 2 8
Example: Find even-numbered nodes

**Depth-First:** 2 4

**Breadth-First:** 2 8
Depth- vs. Breadth- First in Graph

Example: Find even-numbered nodes

Depth-First: 2 4
Breadth-First: 2 8
Example: Find even-numbered nodes

Depth-First: 2 4 8

Breadth-First: 2 8
Depth- vs. Breadth- First in Graph

Example: Find even-numbered nodes

Depth-First: 2 4 8

Breadth-First: 2 8 4
Quiz: Find an Element $> 5$

Depth-First

Breadth-First
Quiz: Identify the Search Order

Depth-First

Breadth-First
Breadth-First “Wave-Front”

d = 0

d = 1

d = 2

d = 3
Stack vs. Queue

Depth-First: Stack: 1

Breadth-First: Queue: 1

DF Order: 1

BF Order: 1
Stack vs. Queue

Depth-First: Stack: 8 2

Breadth-First: Queue: 8 2

DF Order: 1 8

BF Order: 1 2
Stack vs. Queue

Depth-First: Stack: 7 5 2
Breadth-First: Queue: 5 3 8

DF Order: 1 8 7
BF Order: 1 2 8
Stack vs. Queue

Depth-First: Stack: 5 2

Breadth-First: Queue: 7 5 3

DF Order: 1 8 7 5

BF Order: 1 2 8 3
Stack vs. Queue

Depth-First: Stack: 3 2

Breadth-First: Queue: 4 7 5

DF Order: 1 8 7 5 3

BF Order: 1 2 8 3 5
Stack vs. Queue

Depth-First: Stack: 4 2

Breadth-First: Queue: 4 7

DF Order: 1 8 7 5 3 4

BF Order: 1 2 8 3 5 7
Stack vs. Queue

Depth-First: Stack: 2

Breadth-First: Queue: 4

DF Order: 1 8 7 5 3 4 2

BF Order: 1 2 8 3 5 7 4
Stack vs. Queue

Depth-First: Stack: (empty) Breadth-First: Queue: (empty)

DF Order: 1 8 7 5 3 4 2

BF Order: 1 2 8 3 5 7 4
Example: Array ADT

- Domain: Arbitrary
- Associates unique domain element with any index in range 0 to size()-1.
- Size is specified in constructor
- Main Operations:
  - void set(Index, Object)
  - Object get(Index)
  - size()
Classes such as `ArrayList` in Java augment the properties of an array with extendability.

`void add(Object)` adds a new position to the end of an existing array.
**Arrays vs. Lists**

**Complexity-Wise**

- Access time for the $i^{th}$ element of a List is $O(n)$ worst-case, where $n$ is the size of the list. [Slow]

- Access time for the $i^{th}$ element of a List is $O(1)$ worst-case, where $n$ is the size of the array. [Fast]

- (These measures don’t take into account such things as the hidden delay of virtual memory and paging, or the speedup offered by cache memory.)
Why the Disparity?

- Lists use pointers to get to the next element.

- Arrays exploit direct-access capabilities of main memory to get to the location in one memory cycle.
More on Arrays vs. Lists

- Lists are easier to extend element-by-element.

- Closed lists can be used for comparably fast removal of elements.

- Arrays are clumsier in this regard, as the array has to be copied into larger space.
Example: Priority Queue ADT

- Domain: A set $S$ on which there is a linear ordering relation, say $<$

- Operations:
  - void insert($S$)
  - $S$ removeMin()
  - boolean isEmpty()
Implement a Priority Queue

- Requirements \(O(n)\) operation time, where \(n\) is size()
Implement a Priority Queue

- Requirements $O(\log n)$ operation time, where $n$ is $\text{size()}$
Example: Set ADT

- **Domain:** Arbitrary
- **Main Operations:**
  - void add(Object)
  - void remove(Object)
  - boolean isEmpty()
  - int size()
Example: Mapping ADT

- **Domain:** Arbitrary Pairs: $A \times B$ representing a **partial function** $A \rightarrow B$ (partial in that, for a given $a \in A$, there may be no corresponding element of $B$).

- **Main Operations:**
  - `void map(A, B)`
  - `B find(A)`
  - `void unmap(A, B)`
  - `boolean isEmpty()`
  - `int size()`
Example: Relation ADT

- Generalizes Mapping ADT
- Domain: Arbitrary Pairs: $A \times B$
- Main Operations:
  - `void relate(A, B)`
  - `iterator<B> find(A)` // multiple elements
  - `iterator<A> domain()` // multiple elements
  - `void unrelate(A, B)`
  - `boolean isEmpty()`
  - `int size()`