Topic Outline

An Introduction to Stacks and Queues

Stacks

Queue

Representing Stacks and Queues in Memory

Stacks

Queues

The problem of the Crawling Queue

Circular Queues

General Lists

Introduction

Array-based Implementations of Lists

Linked Implementations of Lists

Binary Trees

Introduction

Array-based Implementation of Binary Trees

Linked Implementations of Binary Trees

Notes for Data Structures Lecture

1. **Stack** – LIFO (Last In First Out) – Think of a physical stack of cafeteria trays
   1. Operations
      1. PUSH
      2. POP
   2. Reverses the order of the items.
2. **Queue** – FIFO (First In First Out) – Think of a “Waiting line”
   1. Operations
      1. ENQUEUE
      2. DEQUEUE
   2. Retains items in the order they are entered.

1. Behavior of Stacks in Memory

Operations Stack (Conceptual) Stack (in memory)

Address Value

PUSH “A” Top 🡪 A Top 100 102

101 ?

102 A

103 ?

104 ?

105 ?

Address Value

PUSH “A” Top 🡪 B Top 100 103

PUSH “B” A 101 ?

102 A

103 B

104 ?

105 ?

Address Value

PUSH “A” Top 🡪 C Top 100 104

PUSH “B” B 101 ?

PUSH “C” A 102 A

103 B

104 C

105 ?

Address Value

PUSH “A” Top 🡪 D Top 100 105

PUSH “B” C 101 ?

PUSH “C” B 102 A

PUSH “D” A 103 B

104 C

105 D

1. Behavior of Queues in Memory

Operations Queue (Conceptual) Queue (in memory)

Address Value

ENQUEUE “A” Front / Rear Front 100 102

Rear 101 102

A 102 A

103 ?

104 ?

105 ?

Address Value

ENQUEUE “A” Front / Rear Front 100 102

ENQUEUE “B” Rear 101 103

A B 102 A

103 B

104 ?

105 ?

Address Value

ENQUEUE “A” Front / Rear Front 100 102

ENQUEUE “B” Rear 101 104

ENQUEUE “C” A B C 102 A

103 B

104 C

105 ?

Address Value

ENQUEUE “A” Front / Rear Front 100 102

ENQUEUE “B” Rear 101 105

ENQUEUE “C” A B C D 102 A

ENQUEUE “D” 103 B

104 C

105 D

Additional Concepts:

1. Illustrate how queues “crawl” through memory. Note that stacks don’t crawl.
2. Illustrate a queue collision with another data object.
3. Illustrate the concept of Circular Queues.
4. **Array** – a Data Structure that allows direct access to any individual item (not just “Top” or “Front”) based on an index or subscript.

Operations Array (Conceptual) Array (in memory)

Address Value

Name(1) = “A” Name 100 ?

Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | ? |
| 3 | ? |
| 4 | ? |
| 5 | ? |

Name(2) 102 ?

Name(3) 103 ?

Name(4) 104 ?

Name(5) 105 ?

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | ? |
| 3 | E |
| 4 | ? |
| 5 | ? |

Name(2) 102 ?

Name(3) 103 E

Name(4) 104 ?

Name(5) 105 ?

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | B |
| 3 | E |
| 4 | ? |
| 5 | ? |

Name(2) = “B” Name(2) 102 B

Name(3) 103 E

Name(4) 104 ?

Name(5) 105 ?

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | B |
| 3 | E |
| 4 | L |
| 5 | ? |

Name(2) = “B” Name(2) 102 B

Name(4) = “L” Name(3) 103 E

Name(4) 104 L

Name(5) 105 ?

Items can be inserted in any order. We inserted item 1, then 3, then 2, then 4, in the above example.

The array called “Name” now holds the characters to spell out the name “Abel”.

As with stacks and queues, array items that are conceptually located next to each other (such as the letters “B” and “E” in this example) are stored in adjacent memory locations (locations 102 and 103 in this example).

1. Inserting items into an array – Let’s say we want to change the name stored in the “Name” array from “Abel” to “Mabel” by inserting an “M” at the beginning of the array. You might try simply setting the first element of the array to “M”.

Operations Array (Conceptual) Array (in memory)

Address Value

Name(1) = “M” Name 100 ?

Name(1) 101 M

|  |  |
| --- | --- |
| 1 | M |
| 2 | B |
| 3 | E |
| 4 | L |
| 5 | ? |

Name(2) 102 B

Name(3) 103 E

Name(4) 104 L

Name(5) 105 ?

Unfortunately, that doesn’t produce the desired result as your “M” has overridden your “A”, giving you the non-name “Mbel”.

Instead, what we need to do is to create a ‘hole’ or insertion point at the front of the array so that we can then store the “M” at position 1. In order to do this all of the elements following the insertion point must be moved down (or forward) one position in order to make room for the new element.

Operations Array (Conceptual) Array (in memory)

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | B |
| 3 | E |
| 4 | L |
| 5 | L |

Name(2) = “B” Name(2) 102 B

Name(4) = “L” Name(3) 103 E

--- Name(4) 104 L

Name(5) = Name(4) Name(5) 105 L

Note that I’m copying the last item (the “L” that was in position 4) into the next available slot (position 5). Also note that copying item 4 into position 5 does not erase the previous value in position 4. That is why both Name(4) and Name(5) contain the letter “L” at this point.

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | B |
| 3 | E |
| 4 | E |
| 5 | L |

Name(2) = “B” Name(2) 102 B

Name(4) = “L” Name(3) 103 E

--- Name(4) 104 E

Name(5) = Name(4) Name(5) 105 L

Name(4) = Name(3)

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | B |
| 3 | B |
| 4 | E |
| 5 | L |

Name(2) = “B” Name(2) 102 B

Name(4) = “L” Name(3) 103 B

--- Name(4) 104 E

Name(5) = Name(4) Name(5) 105 L

Name(4) = Name(3)

Name(3) = Name(2)

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 A

|  |  |
| --- | --- |
| 1 | A |
| 2 | A |
| 3 | B |
| 4 | E |
| 5 | L |

Name(2) = “B” Name(2) 102 A

Name(4) = “L” Name(3) 103 B

--- Name(4) 104 E

Name(5) = Name(4) Name(5) 105 L

Name(4) = Name(3)

Name(3) = Name(2)

Name(2) = Name(1)

Finally, now that all of the existing data has been moved down one slot we can insert the letter “M” in the first array position, giving us the name “Mable”.

Address Value

Name(1) = “A” Name 100 ?

Name(3) = “E” Name(1) 101 M

|  |  |
| --- | --- |
| 1 | M |
| 2 | A |
| 3 | B |
| 4 | E |
| 5 | L |

Name(2) = “B” Name(2) 102 A

Name(4) = “L” Name(3) 103 B

--- Name(4) 104 E

Name(5) = Name(4) Name(5) 105 L

Name(4) = Name(3)

Name(3) = Name(2)

Name(2) = Name(1)

Name (1) = “M”

You may be thinking, that was a lot of work to convert “Abel” into “Mabel”, if you wanted the name to be “Mabel” why didn’t you just enter the “M” “A” “B” “E” and “L” in the proper order to begin with? The answer is that I was trying to make the point that inserting items into an existing array can require that you move all of the items after the insertion point down one position (and that you must do these operations in reverse order, from the last towards the first) in order to avoid overwriting existing data.

A more practical application of this concept is when you have an ordered list of items such as student numbers that you want to keep sorted while also being able to add additional student numbers anywhere in the array. When it’s time to add a new student, if you were using an array all of the student numbers following the new student’s number would need to be moved to make a slot for the new student.

1. Revisiting Selection Sort now that we’ve learned about arrays. Selection Sort works by repeatedly removing the smallest item from an unsorted list and appending that item to the end of a sorted list. Conceptually, there are two lists, the sorted list and the unsorted list. Typically Selection Sort and many other sorts are implemented using a single array to hold both the sorted and unsorted portions of the list. In order to minimized the number of items that need to be moved a system for ‘swapping’ items is typically employed.

To begin with, at the start of the process the entire list is assumed to be unsorted. The list is searched for the smallest item and once found that smallest item is copied to the first position in the array and the item previously held in the first position is placed into the position formerly held by the smallest item. In order to accomplish this “swap” a “temporary variable” is needed to hold a copy of the first item so that its value is not forgotten when the smallest item is written into the first slot.

The process repeats with the items in the array from slot 2 through the end of the list searched for the smallest item and that item is then swapped with the item in position 2. The process is repeated for positions 3, 4, 5 etc. until the entire list is sorted. This is illustrated below.

Conceptual “in place” implementation using an array

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| E | A | D | C | B |
| 1 | 2 | 3 | 4 | 5 |

E A D C B

Sorted Unsorted Sorted Unsorted Swap code

Temp = List(1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | E | D | C | B |
| 1 | 2 | 3 | 4 | 5 |

E D C B

A

List(1) = List(2)

List(2) = Temp

Temp = List(2)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | D | C | E |
| 1 | 2 | 3 | 4 | 5 |

E D C

A B

List(2) = List(5)

List(5) = Temp

E D

A B C

Temp = List(3)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | E |
| 1 | 2 | 3 | 4 | 5 |

List(3) = List(4)

List(4) = Temp

E

A B C D

Temp = List(4)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | E |
| 1 | 2 | 3 | 4 | 5 |

List(4) = List(4)

List(4) = Temp

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | E |
| 1 | 2 | 3 | 4 | 5 |

A B C D E

1. **Linked Lists** – A data structure that implements a list of items as a collection of elements where each element includes both a value and the location of the next item in the list. Logically adjacent items do *not* need to be in physically adjacent memory locations in a liked list structure.

Linked List (in memory)

Start

|  |
| --- |
| 107 |

100

Address Value

Start 100 107

|  |  |
| --- | --- |
| A | 103 |
| 107 | 108 |

101 E

|  |  |
| --- | --- |
| C | 109 |
| 103 | 104 |

102 -1

103 C

104 109

105 ?

106 ?

|  |  |
| --- | --- |
| E | -1 |
| 101 | 102 |

|  |  |
| --- | --- |
| D | 101 |
| 109 | 110 |

107 A

108 103

109 D

110 101

So far we have maintained the order of items in a list by arranging items in memory to reflect their logical order. Another way of organizing a list is to have each item in the list link (or point) to the location of the next item in the list. One reason for doing this is so that we can insert items into the middle of a list without needing to move all of the items below the insert point.

Linked List (in memory)

Start

|  |
| --- |
| 107 |

100

Address Value

Start 100 107

|  |  |
| --- | --- |
| A | 105 |
| 107 | 108 |

101 E

|  |  |
| --- | --- |
| C | 109 |
| 103 | 104 |

102 -1

103 C

104 109

105 B

106 103

|  |  |
| --- | --- |
| E | -1 |
| 101 | 102 |

|  |  |
| --- | --- |
| D | 101 |
| 109 | 110 |

107 A

108 105

109 D

|  |  |
| --- | --- |
| B | 103 |
| 105 | 106 |

110 101

1. **Binary Trees** –

Conceptual organization

Root 🡪

Leaf 🡪 🡨 Leaf

How are binary trees used?

1. Binary trees are often used to structure data in ways that facilitate rapid search. For example, we may organize the alphabetic data, such as names, so that all elements preceding the current element are to the left of that element and all elements following the current element are to the right of that element. Thus, since “LeMoine” precedes “Ourso” it is to the left while “Tarver” which follows “Ourso” alphabetically is to the right. Similarly, “Bader” is to the left of “LeMoine” and “Losito” to the right.

Binary trees can also form the basis of more complex data structures, such as “Red / Black Trees”, that support both rapid retrieval and quick insertion and deletion of data.

1. Binary trees are also frequently used to hold mathematical expressions so that they can be evaluated efficiently.

For example, the mathematical expression “= 5 \* 2 + 3” will add the product of 5 \* 2 (which is 10) to 3 giving 13. [Remember that multiplication has a higher precedence than addition so the multiplication must the performed first.] The structure of this expression can be captured in binary tree form as follows:

Evaluation takes places in a bottom up fashion, with the multiplication of 5 and 2 happening before the addition of 3. If this seems odd at first, think about the fact that you won’t know what to add the 3 to unless you first do the multiplication of 5 and 2.

Of course, it is possible to force addition to take place before multiplication by using parentheses. Thus, the mathematical expression “= 5 \* ( 2 + 3 )” will multiple 5 by the sum of 2 + 3 (which is also 5) giving 25. The structure of this expression can be expressed by the following binary tree.

One final thing to notice about these two binary trees is that they are “unbalanced”. A perfectly balanced binary tree will be symmetric about its root element. These trees are lopsided with either more nodes to the left of the root or more nodes to the right of the root.

Binary tree implementation using arrays

As odd as it may sound, a binary tree can be stored in an array. When doing so we use three simple mathematical formulas to specify the location of a node’s left child, right child, and parent (if they exist). These formulas are:

* The left child of node i will be located at 2i LeftChild(i) = 2 \* i
* The right child of node i will be located at 2i+1 RightChild(i) = (2 \* i) + 1
* The parent of node i will be located at Parent(i) = Floor(i/2)

Root 🡪

Leaf 🡪 🡨 Leaf

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Level 1 | Level 2 | | Level 3 | | | |
| A | B | C | D | E | F | G |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Thus, the element “C” which is located at position 3 has it’s left child “F” stored at position 6 (because 2 \* 3 = 6) and it’s right child “G” stored at position 7 (since 2 \* 3 + 1 = 6 + 1 = 7). The parent of “C” is located at position 1 (since 3/2 = 1.5 and the floor of 1.5 is 1. Note that “floor” means to drop any fraction.)

The parent of element “F” is located at position 3 because “F” is at position 6 and 6/2 = 3.0 = 3. Thus, “C” is “F’s” parent. Note that if “F” had a RIGHT child that child would be located at position 13 (since “F” is at position 6 and 2 \* 6 + 1 = 12 + 1 = 13).

The root node “A” is the only node in the tree that does not have a parent. If you apply the parent position formula to “A” which is at location 1, you get zero (since ½ = 0.5 and the floor of 0.5 is 0) which is outside the array, as the first element is at position 1.

Once you know the formulas the array implementation of a tree is easy to understand and implement. However, this implementation approach does suffer from one serious weakness.

Only when a binary tree is both balanced and full does the array implementation utilize space efficiently. When binary trees are unbalanced or less than full the array implementation uses memory inefficiently.

For example, here is a binary tree representing the expression “= 5 \* (2 + 3)” and its implementation using an array.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Level 1 | Level 2 | | Level 3 | | | |
| \* | 5 | + |  |  | 2 | 3 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Note that two of the seven array locations are ‘unused”. Wasting two locations may not seem like a big deal, but given that the array only has seven total slots this represents wasting 28.6% (2 slots / seven slots) – between a quarter and a third of total space. For larger trees that are even less balanced the percentage of wasted space can grow quite large.

As mentioned above, the array implementation uses space efficiently only with the binary tree is both “balanced” and “full”. **Full** means that every node that is not a leaf has exactly two children. Leaves, by definition have no children. **Balanced** means the tree is symmetric about its root node. It is possible for a tree to be “balanced” but not “full” as illustrated below.

Root 🡪

Leaf 🡪 🡨 Leaf

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Level 1 | Level 2 | | Level 3 | | | |
| A | B | C | D |  |  | G |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

For binary trees that are highly unbalanced and / or sparse, the amount of space wasted in an array implementation of a binary tree can be extreme.

Root 🡪

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lvl 1 | Level 2 | | Level 3 | | | | Level 4 | | | | | | | |
| A | B | C |  | D |  | E |  |  | F |  |  |  |  | G |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

The above example wastes over ½ the total space reserved for the array (53.3% since 8 empty slots / 15 total slots = 0.5333). Just imagine how much space would be wasted if G had a right child “H” and no other changes were made to the tree. [Well, don’t imagine, let’s work it out. “G” is at position 15 so its right child would be at position 31 and only 8 out of 31 slots would be filled (A, B, C, D, E, F, G, H; 8 items), meaning 25.8% of the space would be used (8/31 = 0.258) and a whopping 74.2% wasted.]

Binary tree implementation using pointers

Root 🡪

Leaf 🡪 🡨 Leaf

Start

|  |
| --- |
| 107 |

100

Binary Tree (in memory)

Address Value

Start 100 107

|  |  |  |
| --- | --- | --- |
| A | 104 | 101 |
| 107 | 108 | 109 |

101 C

102 110

103 116

104 B

|  |  |  |
| --- | --- | --- |
| B | 119 | 113 |
| 104 | 105 | 106 |

105 119

|  |  |  |
| --- | --- | --- |
| C | 110 | 116 |
| 101 | 102 | 103 |

106 113

107 A

108 101

|  |  |  |
| --- | --- | --- |
| D | -1 | -1 |
| 119 | 120 | 121 |

109 104

|  |  |  |
| --- | --- | --- |
| G | -1 | -1 |
| 116 | 117 | 118 |

|  |  |  |
| --- | --- | --- |
| E | -1 | -1 |
| 113 | 114 | 115 |

|  |  |  |
| --- | --- | --- |
| F | -1 | -1 |
| 110 | 111 | 112 |

110 F

111 -1

112 -1

113 E

114 -1

115 -1

116 G

117 -1

118 -1

119 D

120 -1

121 -1

Note that this example gives a somewhat skewed impression of space utilization. Generally the “data” portion of tree nodes are much larger than the “pointer” portions of tree nodes so, in general, pointer based trees are much more efficient in terms of space utilization than they appear here and are, in fact, far more space efficient than array based binary trees.