

CS 1332 - Week 03

No Content - Monday

Wednesday

Topics

- I. Stacks review
 - A. Stack ADT - abstract data type with the expected “Last In, First Out” (LIFO) behavior; can be implemented with an Array, SLL, or DLL
- II. Queues
 - A. An ADT like the stack, but with the expected “First In, First Out” (FIFO) behavior.
 - B. Queues are very linear structures
 - 1. Applications: a water pipe, waitlists, print queues, online orders
 - C. Operations:
 - 1. enqueue(x) - adds to one end
 - 2. x dequeue() - removes from the opposite end
 - 3. x peek() - returns the item at the front of the queue without removing it
 - 4. isEmpty()
 - 5. size()
- III. SLL-backed queue
 - A. Must have a head and a tail - which end do we perform each operation at?
 - 1. If we add to the head, we have to remove from the tail, but removing from the tail of a SLL is $O(n)$
 - 2. Instead, add to the tail and remove from the head
 - a) Adding to the tail - $O(1)$
 - b) Removing from the head - $O(1)$
- IV. Array-backed queue
 - A. Values we will keep track of:
 - 1. capacity - length of the backing array
 - 2. front index - of first element
 - 3. back index - of last element (can also be the spot after the last el.)
 - 4. size - number of elements actually present in the array
 - B. Example 1:
 - 1. enqueue “ramblin” (ignoring resizing for now)

0	1	2	3	4	5	6
r	a	m	b	l	i	n
Front						Back

character	size	front	back
r	1	0	0
a	2	0	1
m	3	0	2
b	4	0	3
l	5	0	4
i	6	0	5
n	7	0	6

2. dequeue 4 times

character	size	front	back
r	6	1	6
a	5	2	6
m	4	3	6
b	3	4	6

3. enqueue "wr"

character	size	front	back
w	4	4	0 (7 % 7)
r	5	4	1

a) Mod the back by capacity when adding to the back

b) Mod the front by capacity when removing

V. Deques

A. Deque - "Double Ended Queue," does not have a LIFO or FIFO behavior

1. Defined by ability to perform add/remove operations from both ends of the structure

B. Operations:

1. addFirst(x)
2. addLast(x)
3. x removeFirst()
4. x removeLast()

- VI. DLL-backed deque
 - A. addFirst() - add at the head
 - B. addLast() - add at the tail
 - C. removeFirst() - remove from the head
 - D. removeLast() - remove from the tail
 - E. We can add and remove from the front and back in $O(1)$ time
- VII. Array-backed deque
 - A. Uses the same "circular" array we used when creating an Array-backed queue
 - B. addFirst() - add at the front index
 - 1. front should be set to capacity - 1 if decremented to -1
 - C. addLast() - add at the back index
 - 1. back should be mod by capacity when incrementing
 - D. removeFirst() - remove from the front index
 - 1. front should be mod by capacity when incrementing
 - E. removeLast() - remove from the back index, make sure
 - 1. back should be set to capacity - 1 if decremented to -1

Friday

Topics

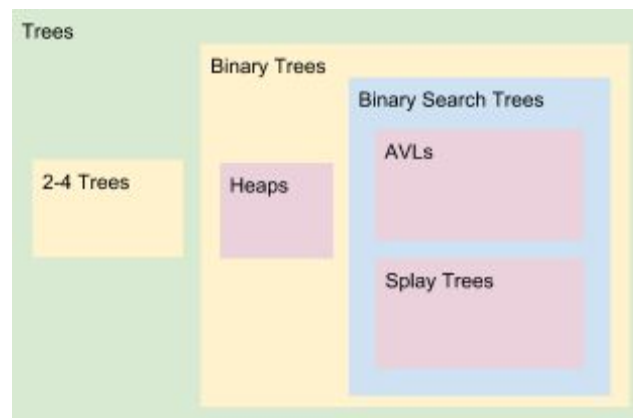
- I. Overview of worst case runtimes

	Access - known index	Search - unknown index
Array	$O(1)$	$O(n)$
SLL	$O(n)$	$O(n)$
DLL	$O(n)$	$O(n)$
Stack	$O(n)$	$O(n)$
Queue	$O(n)$	$O(n)$
Deque	$O(n)$	$O(n)$

- A. The data structures that access and search in $O(n)$ are not bad data structures because they are simply not built for this purpose
- II. Trees Intro
 - A. Characteristics:
 - 1. No cycles - you cannot reach a node from itself (eg. circular SLLs have a cycle, non-circular SLLs do not have any cycles)
 - 2. Highly recursive
 - 3. ADT - can be implemented with Arrays (messy in most cases) and Linked List-like structures with nodes and pointers (preferred)

B. Properties - types of trees are defined by their *shape* and *order* properties

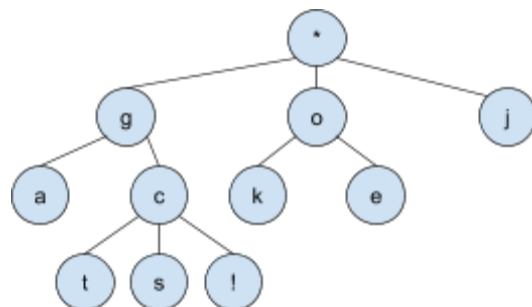
1. Shape - structure of the tree
2. Order - arrangement of data in the structure



C. Terminology:

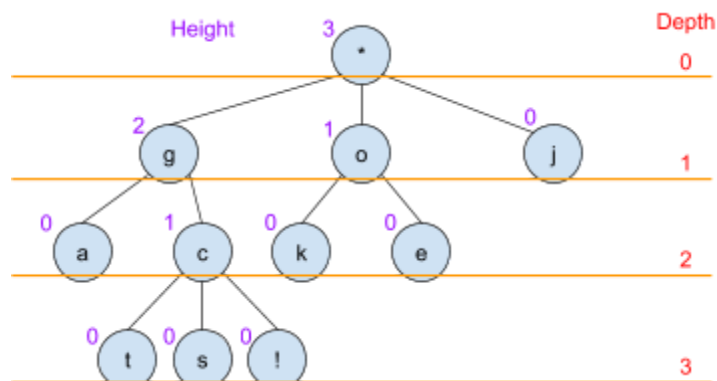
1. Root - like the head of a LL; the entry point for the tree
2. Children - nodes of a SLL have one “child”; tree nodes can have more
3. External/Leaf nodes - nodes without children
4. Internal nodes - nodes with children

D. Hierarchy:



1. g is the *parent* of a and c and *grandparent* of t and s
2. a and c are *siblings*
3. a and c are *cousins* of k and e
4. c,t,s,l is a *subtree* of g

E. Depth & Height



1. Depth - distance of a node from the root
 - a) Depth of the root is always 0
2. Height - distance of a node from the furthest leaf node
 - a) Height of a leaf node is always 0
 - b) The height of a node can be calculated by adding 1 to the maximum height of its children
 - (1) Eg. $\text{height}(g) = 1 + \max(\text{height}(a), \text{height}(c)) = 1 + 1$

F. Tree ADT operations

1. Information methods:
 - a) `size()`
 - b) `isEmpty()`
 - c) `iterator()`
 - d) `position()` - returns a list of all node positions
2. Accessor methods:
 - a) `root()` - returns root of the tree
 - b) `parent(x)` - returns the parent of node x
 - c) `children(x)` - returns the children of node x
 - d) `numChild(x)` - returns the number of children node x has
3. Query methods:
 - a) `isInternal()` - called on a node
 - b) `isExternal()` - called on a node
 - c) `isRoot()` - called on a node

III. Binary Trees

- A. Shape - each node can have at most 2 children
 1. Node stores at minimum child references and data but can also contain:
 - a) root - reference to the root
 - b) parent - reference to its parent
 - c) internal - is it an internal node
 - d) external - is it an external node
- B. Iterating through a binary tree
 1. Use recursion to iterate while the current node is not null, this allows you to access parent nodes when you begin returning from the recursive calls