Data Structures and Algorithm Analysis (CSC317)

Intro/Review of Data Structures
Focus on dynamic sets

We’ve been talking a lot about efficiency in computing and run time

…. But thus far mostly ignoring data structures

Dynamic sets …

- Set size changes over time
- Elements could have identifying keys, and could also have satellite data

example: key corresponding to friend name, with satellite data corresponding to email, phone, favorite hobbies, etc

Dynamic sets … what operations?

Dynamic sets … what operations?

- Either **queries**
- Or **modifying operations** that change the set

Dynamic sets … what operations?

- Search
- Insert
- Delete
- Min / Max
- Successor / Predecessor

Operations on dynamic sets…

Which data structure?

- Depends on what you want to do.

We know of… ?

Operations on dynamic sets…

Which data structure?

- Depends on what you want to do.

We know of… hash table, stack, queue, linked list, tree, heap, etc.

Data structures

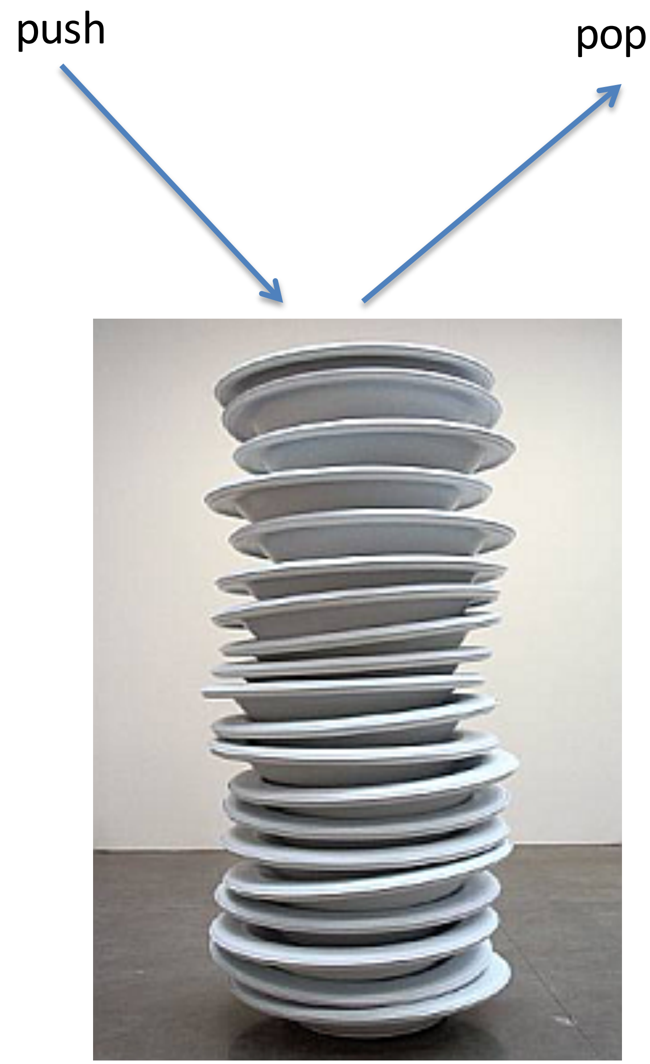
Hash table

- Insert, Delete, Search/lookup
- We ***don't*** maintain order information
- Applications?
- We'll go through in detail later
- We'll see that all operations **on average O(1)**

Data structures

Stack

- last-in-first-out
- Insert = push
- Delete = pop
- Applications?



Data structures

Stack

Run time of push and pop? O(1)

Very fast!

But limited operations… (eg, if you want to Search it's not efficient)

Data structures

Queue

- first-in-first-out

Data structures

Queue

- first-in-first-out
- Insert = Enqueue
- Delete = Deqeue
- Applications?

Data structures

Queue

- first-in-first-out

Run time Enqueue/Dequeue: O(1)

Very fast!

But limited operations…

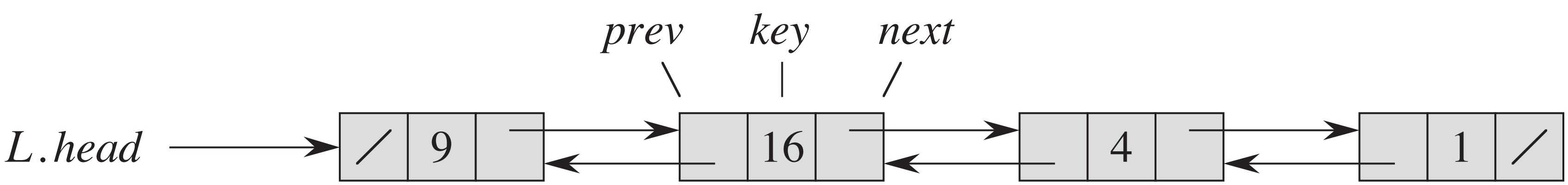
Data structures

Linked lists

- Search
- Insert
- Delete

Data structures

Linked lists (example of double linked)



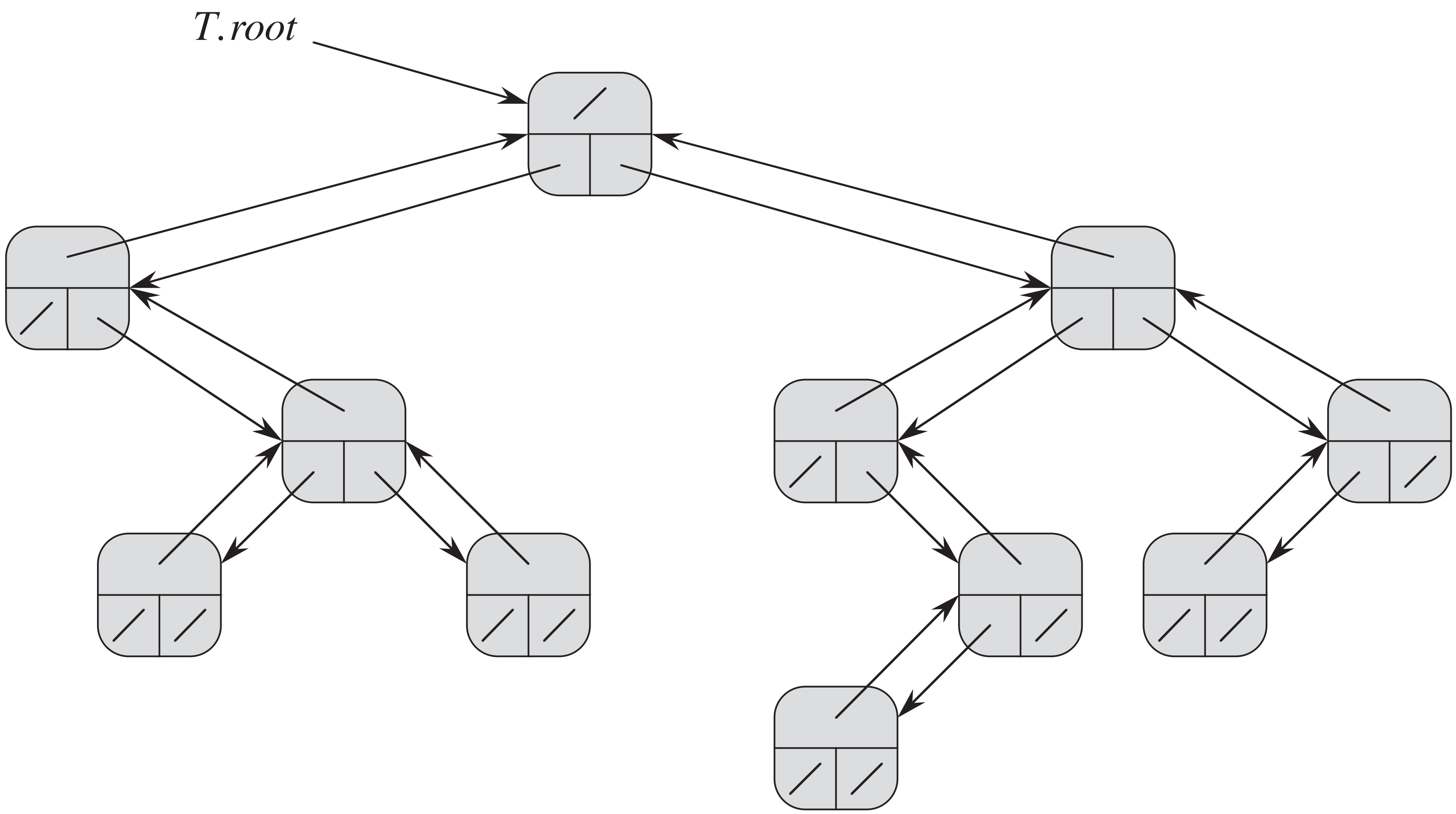
Data structures

Linked lists: Run time?

- Search O(n) [limitation if lots of searches]
- Insert O(1)
- Delete O(1) [unless first searching for key]

Data structures

Binary tree and trees (later)



Data structures

Binary trees

- Search
- Min/Max
- Predecessor/Successor
- Insert/Delete
- Later; basic operations take height of tree, complete binary tree $\Theta(\log n)$

Data structures

Heap: main operations: (discussed in sorting chapter)

- insert $\Theta(\log n)$
- Remove object from heap that is min (*or* max, *but not both*) $\Theta(\log n)$
- Technically, can be implemented via a complete binary tree
- Applications?

Data structures

Heap: main operations: (discussed in sorting chapter)

- insert $\Theta(\log n)$
- Remove object from heap that is min (*or* max, but not both) $\Theta(\log n)$
- Applications?
- (Heapsort) and we'll discuss finding median dynamically…

Finding median dynamically

Input: numbers presented one by one: $x_1, x_2, \ldots x_n$

Output: At each time step, the median

Run time?

Finding median dynamically

Input: numbers presented one by one: $x_1, x_2, \ldots x_n$

Output: At each time step, the median

Run time? We know we can do O(n) but dynamically each time we add a number, would like to do better and not have to recompute with O(n)

Finding median dynamically

Input: numbers presented one by one: $x_1, x_2, \ldots x_n$

Output: At each time step, the median

- Using two heaps: one for max and one for min
  O(log k) each step

On the board…

Finding median dynamically

Low Heap holding smaller numbers: performs **max** operation in O(log k) time

High Heap holding larger numbers: performs **min** operation in O(log k) time

Invariant: half smallest number of elements so far in low heap; half highest in high heap

Finding median dynamically

Low Heap (max); High heap (min)

Invariant: half smallest number of elements so far in low heap; half highest in high heap

- Consider if have 10 elements and inserting the 11th; 12th - need to maintain balanced number in each heap
- If Low has 6 elements and High 5 elements, and next element is less than max of Low, insert in low and move min of High to Low…

Finding median dynamically

Low Heap (max); High heap (min)

Computing median: each step log(k) time

- If k is odd number (eg, 6 in Low and 5 in High), extract min of High
- If k is odd number (eg, 5 in Low and 6 in High), extract max of Low
- If k even number, extract both min of High and max of Low