Lectures 3, 4
Recap of basic data structures, binary search

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Outline of these notes

- Review of basic data structures
- Searching in a sorted array/binary search: the algorithm, analysis

Basic Data structures

Prerequisite material. Review [GT Chapters 2–4, 6] as necessary)

- Arrays, dynamic arrays
- Linked lists
- Stacks, queues
- Dictionaries, hash tables
- Binary trees

Arrays, Dynamic arrays, Linked lists

- Arrays:
  - Numbered collection of cells or entries
    - Numbering usually starts at 0
    - Fixed number of entries
  - Each cell has an index which uniquely identifies it.
  - Accessing or modifying the contents of a cell given its index: $O(1)$ time.
  - Inserting or deleting an item in the middle of an array is slow.
- Dynamic arrays:
  - Similar to arrays, but size can be increased or decreased
  - `ArrayList` in Java, `list` in Python
- Linked lists:
  - Collection of nodes that form a linear ordering.
    - The list has a first node and a last node
    - Each node has a next node and a previous node (possibly null)
  - Inserting or deleting an item in the middle of linked list is fast.
  - Accessing a cell given its index (i.e., finding the kth item in the list) is slow.

Stacks and Queues

- Stacks:
  - Container of objects that are inserted and removed according to Last-In First-Out (LIFO) principle:
    - Only the most-recently inserted object can be removed.
  - Insert and remove are usually called `push` and `pop`
- Queues (often called FIFO Queues)
  - Container of objects that are inserted and removed according to First-In First-Out (FIFO) principle:
    - Only the element that has been in the queue the longes can be removed.
  - Insert and remove are usually called `enqueue` and `dequeue`
  - Elements are inserted at the rear of the queue and are removed from the front

Dictionaries/Maps

- Dictionaries
  - A Dictionary (or Map) stores `<key,value>` pairs, which are often referred to as items
  - There can be at most item with a given key.
  - Examples:
    1. `<Student ID, Student data>`
    2. `<Object ID, Object data>`

Hashing

An efficient method for implementing a dictionary. Uses

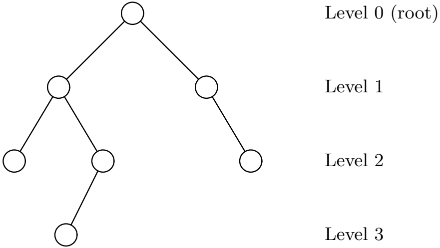
- A hash table, an array of size N.
- A hash function, which maps any key from the set of possible keys to an integer in the range $[0, N-1]$
- A collision strategy, which determines what to do when two keys are mapped to the same table location by the hash function.Commonly used collision strategies are:
  - Chaining
  - Open addressing: linear probing, quadratic probing, double hashing
  - Cuckoo hashing

Hashing is fast:

- $O(1)$ expected time for access, insertion
- Cuckoo hashing improves the access time to $O(1)$ worst-case time. Insertion time remains $O(1)$ expected time.

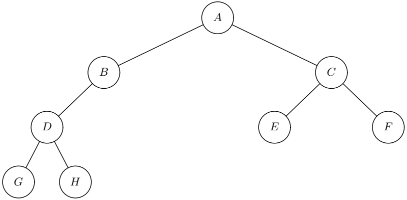
Binary Trees: a quick review

We will use as a data structure and as a tool for analyzing algorithms.



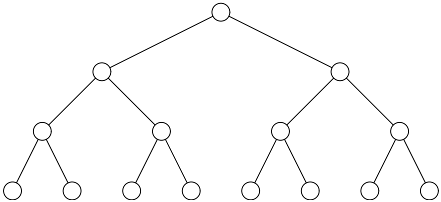
The depth of a binary tree is the maximum of the levels of all its leaves.

Traversing binary trees



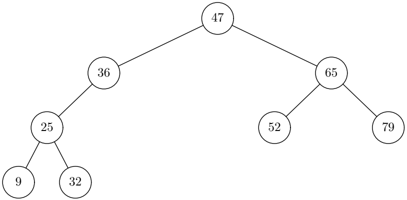
- Preorder: root, left subtree (in preorder), right subtree (in preorder): *ABDGHCEF*
- Inorder: left subtree (in inorder), root, right subtree (in inorder): *GDHBAECF*
- Postorder: left subtree (in postorder), right subtree (in postorder), root: *GHDBEFCA*
- Breadth-first order (level order): level 0 left-to-right, then level 1 left-to-right, . . . : *ABCDEFGH*

Facts about binary trees



1. There are at most 2^k nodes at level k.
2. A binary tree with depth d has:
   - At most 2^d leaves.
   - At most $2^{d+1} - 1$ nodes.
3. A binary tree with n leaves has depth $\geq \lceil \lg n \rceil$.
4. A binary tree with n nodes has depth $\geq \lfloor \lg n \rfloor$.

Binary search trees



- Function as ordered dictionaries. (Can find successors, predecessors)
- find, insert, and remove can all be done in $O(h)$ time ($h =$ tree height)
- AVL trees, Red-Black Trees, Weak AVL trees: $h = O(\log n)$, so find, insert, and remove can all be done in $O(\log n)$ time.
- Splay trees and Skip Lists: alternatives to balanced trees
- Can traverse the tree and list all items in $O(n)$ time.
- [GT] Chapters 3–4 for details

Binary Search: Searching in a sorted array

- Input is a sorted array A and an item x.
- Problem is to locate x in the array.
- Several variants of the problem, for example...
  1. Determine whether x is stored in the array
  2. Find the largest i such that $A[i] \leq x$ (with a reasonable convention if $x < A[0]$).

  We will focus on the first variant.
- We will show that binary search is an optimal algorithm for solving this problem.

Binary Search: Searching in a sorted array

Input: A: Sorted array with n entries $[0..n-1]$
x: Item we are seeking

Output: Location of x, if x found
-1, if x not found

```
def binarySearch(A,x,first,last)
if first > last:
  return (-1)
else:
  mid = ⌊(first+last)/2⌋
  if x == A[mid]:
    return mid
  else if x < A[mid]:
    return binarySearch(A,x,first,mid-1)
  else:
    return binarySearch(A,x,mid+1,last)

binarySearch(A,x,0,n-1)
```

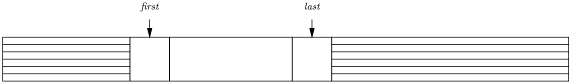
Correctness of Binary Search

We need to prove two things:

1. If x is in the array, its location in the array (its index) is between *first* and *last*, inclusive.
   Note that this is equivalent to:

   > *Either x is not in the array, or its location is between first and last, inclusive.*

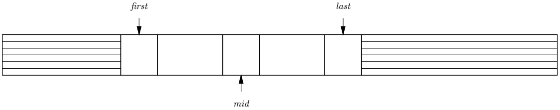
2. On each recursive call, the difference *last* − *first* gets strictly smaller.



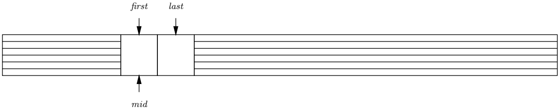
Correctness of Binary Search

To prove that the invariant continues to hold, we need to consider three cases.

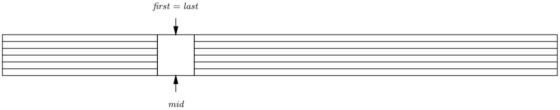
1. $last \geq first + 2$



2. $last = first + 1$



3. $last = first$



Binary Search: Analysis of Running Time

- We will count the number of 3-way comparisons of x against elements of A. (also known as decisions)
- Rationale:
  1. This is the essentially the same as the number of recursive calls. Every recursive call, except for possibly the very last one, results in a 3-way comparison.
  2. Gives us a way to compare binary search against other algorithms that solve the same problem: searching for an item in an array by comparing the item against array entries.

Binary Search: Analysis of Running Time (continued)

- Binary search in an array of size 1: 1 decision
- Binary search in an array of size $n > 1$: after 1 decision, either we are done, or the problem is reduced to binary search in a subarray with a worst-case size of $\lfloor n/2 \rfloor$
- So the worst-case time to do binary search on an array of size n is $T(n)$, where $T(n)$ satisfies the equation

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases}$$

- The solution to this equation is:

$$T(n) = \lfloor \lg n \rfloor + 1$$

  This can be proved by induction.
- So binary search does $\lfloor \lg n \rfloor + 1$ 3-way comparisons on an array of size n, in the worst case.