DS286 | 2016-11-07

Midterm Solutions

Yogesh Simmhan
simmhan@cds.iisc.ac.in

Delete from Sorted Doubly Linked List

```
Node head;
void delete(int val) {
   Node curr = head;
   while(curr != null && curr.item < val) curr = curr.next;
   if(curr == null || curr.item > val) return; // Not found
   if(curr.prev == null) head = curr.next; // Delete head
   else curr.prev.next = curr.next;      // Delete internal
   if(curr.next != null) curr.next.prev = curr.prev;
   delete curr;
}
```

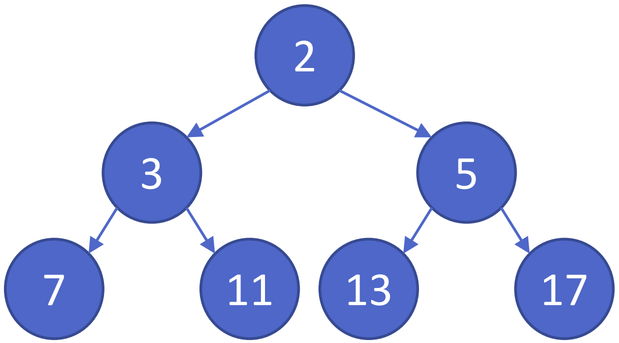
Dictionary search using sorted array

```
int search(int key, Pair[] slist, int s, int e) {
   int match = -1;
   if (e < s) return match;
   int mid = (s+e)/2;
   if (slist[mid].key == key) return slist[mid].val;
   else
      if (key < slist[mid].key)
         return search(key, slist, s, mid-1);
      else    // key > slist[mid].key
         return search(key, slist, mid+1, e);
}
```

Dictionary search complexity

- Best case: O(1)
- Worst case: O(log n)
- Expected case: O(log n)

Full Binary Tree of Primes

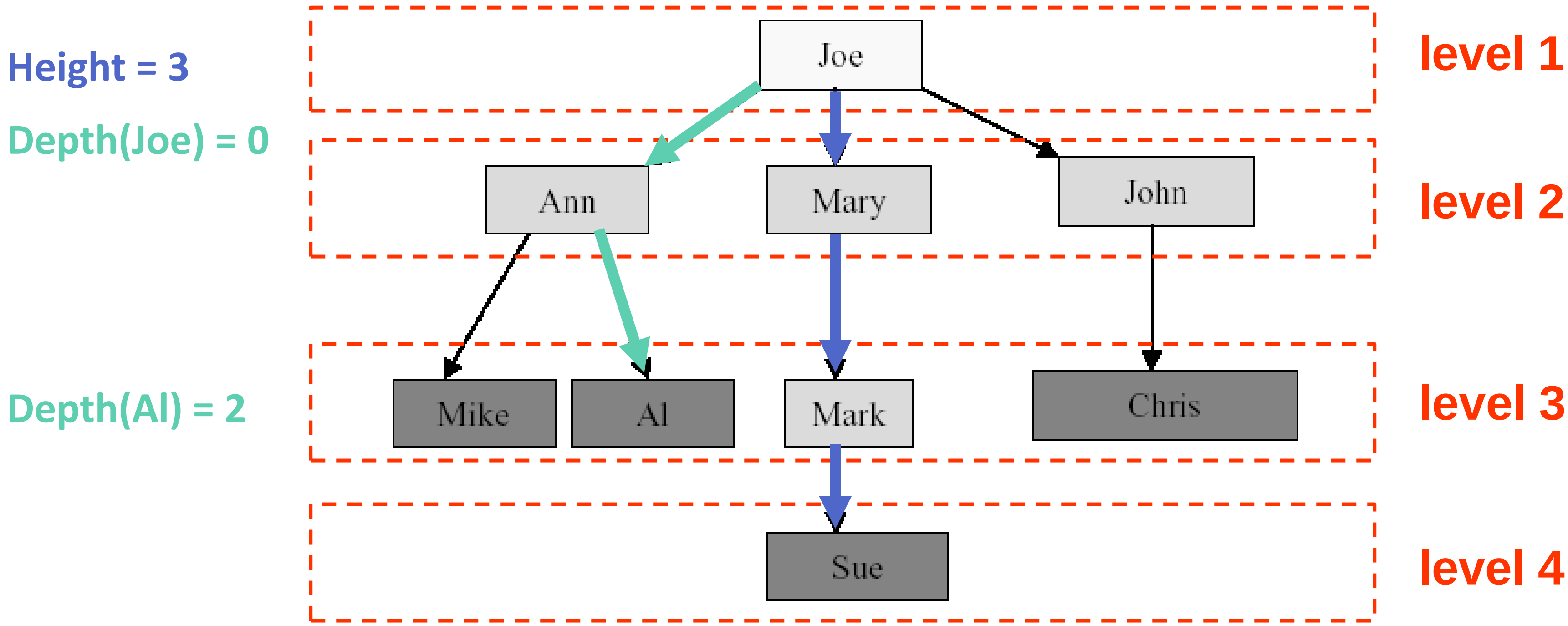


Levels = 3
Height = 2
(But assuming 3 will be graded since listed in slides)

- Inorder: 7, 3, 11, 2, 13, 5, 17
- Preorder: 2, 3, 7, 11, 5, 13, 17
- Postorder: 7, 11, 3, 13, 17, 5, 2

Levels and Height

- **Depth** of a Node = Number of edges from the root to that node
- **Height** of a Tree = Number of edges from root to farthest leaf, i.e. Max(depth) over all leaves
- Number of **Levels** of a Tree = Height + 1



Binary Tree Properties

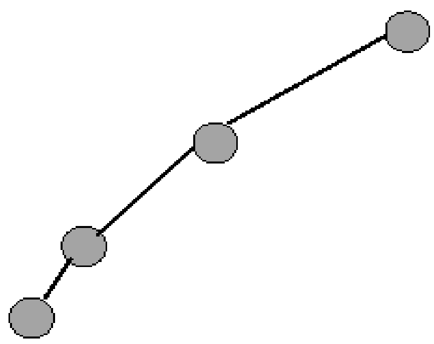
1. The drawing of every binary tree with n elements, $n > 0$, has exactly n-1 edges.
   – Each node has exactly 1 parent (except root)
2. A binary tree of height h, $h >= 0$, has at least $h+1$ and at most $2^{h+1}-1$ elements in it.
   - h+1 levels; at least 1 element at each level → #elements = h+1
   - At most 2^{i-1} elements at i-th level → $\Sigma\, 2^{i-1} = 2^{h+1} - 1$

$$a+ar^1+ar^2+\ldots+ ar^n = a(r^{n+1}-1)/(r-1)$$

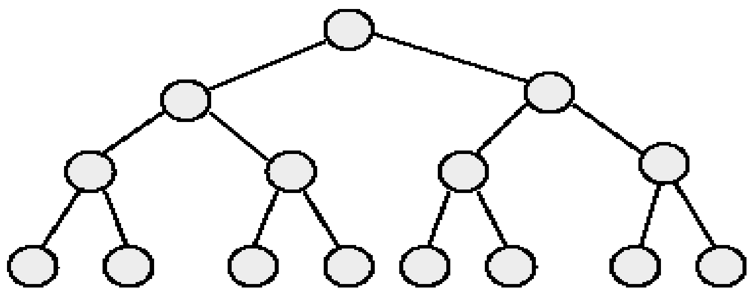
Note: Some tree definitions differ between computer science & discrete math

Binary Tree Properties

3. The height of a binary tree that contains *n* elements, *n* >= 0, is at least $\lfloor \log_2 n \rfloor$ and at most *n-1*.
   - At least one element at each level → h_{max} = #elements - 1
   - From prev: h_{min} = ceil(log(n+1))



minimum number of elements

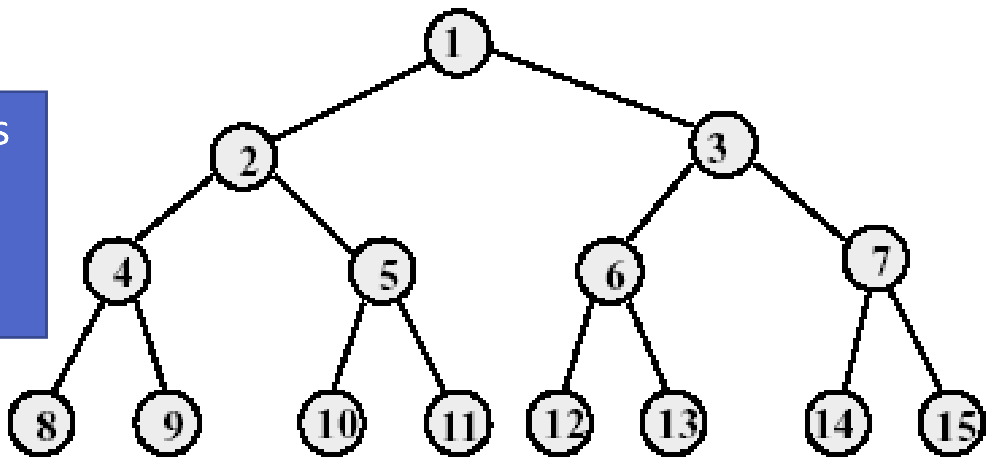


maximum number of elements

Full Binary Tree

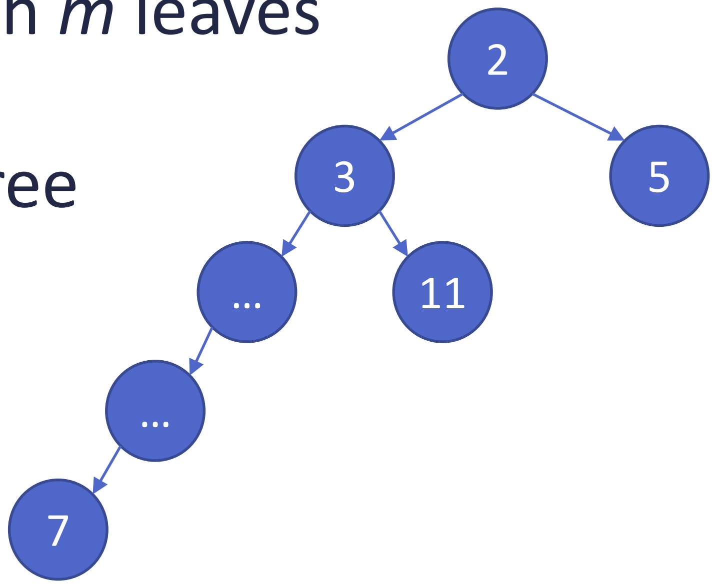
- A full binary tree of height h has exactly $2^{h+1}-1$ nodes
- Numbering the nodes in a full binary tree
  - Number the nodes 1 through $2^{h+1}-1$
  - Number by levels from top to bottom
  - Within a level, number from left to right

Note: Some definitions of full, complete trees are NOT consistently used everywhere



Tree height and nodes

- Maximum nodes in binary tree with *m* leaves
  - **Infinity!**
- But, if assuming "Proper" Binary tree
  - i.e. every node has 0 or 2 children
    - Every pair of leaf has 1 parent
    - Every internal node pair has 1 parent
  - m+m/2+m/4+…+1=**2m-1**
  - *Does not have to be full/complete*

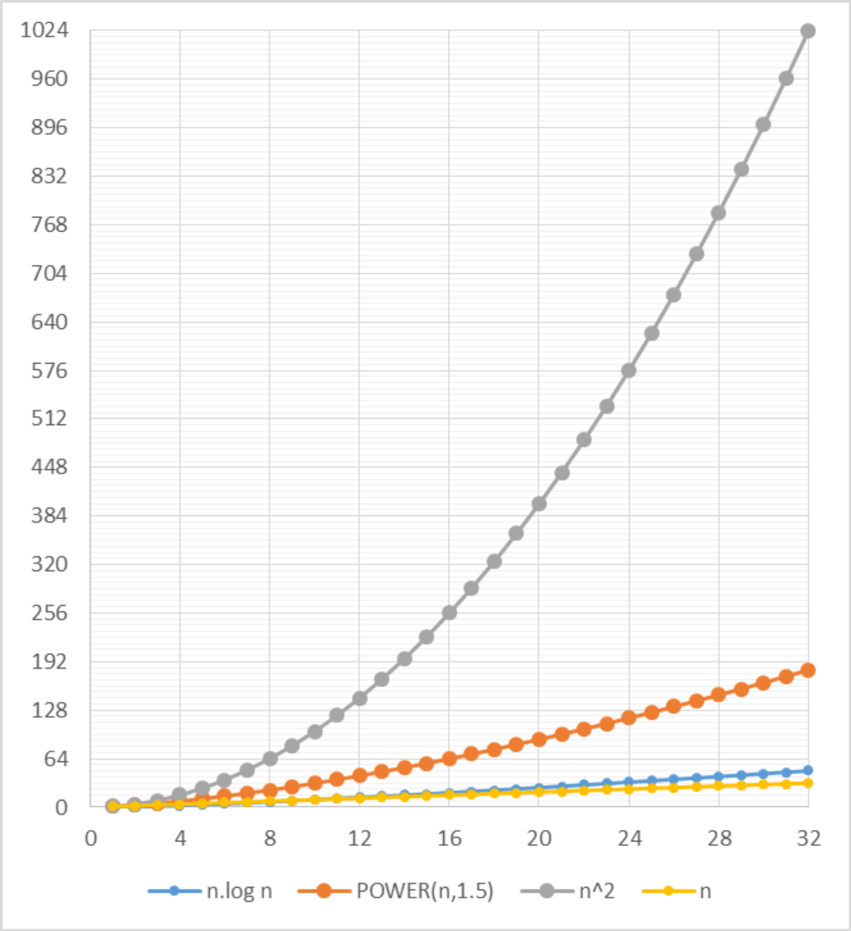


- Minimum height of binary tree with *n* nodes
  - Minimum height when it is complete
  - $\lfloor \log_2 n \rfloor$
  - *Any reasonable answer is given full points for grading.*

Basket: Insert, lookup

- BigBasket
  - Space: $O(n)$ Time: **insert()** = $O(n^2)$, $\Omega(n.\log n)$; **lookup()** = $O(n.\log n)$
  - Takes less space, suitable for storing large number of items in memory
  - Insertion time upper bound is very high and lower bound is low. Large variability between upper and lower bounds
  - Lookup time upper bound is medium.
  - Well suited when large number of items have to be stored in the ADT, with few insertions but with many lookups that take medium latency.
- FastBasket
  - Space: $O(n^2)$ Time: **insert()** = $\Theta(n^{1.5})$; **lookup()** = $O(n)$
  - Takes a lot of space and is not suited for storing large number of items
  - Insertion time is medium, but it is a tight bound. So good for frequent insertions with deterministic time bound if size does not grow large (need to delete)
  - Lookup time upper bound is low, so good for frequent lookups as well.
  - Well suited for applications with frequent insertions and lookups with low latency, as long as total size does not grow large and fits within memory.

Complexity



- Specific values of *n* do not make things good or bad, e.g. n=1000 may be horrible for O(n^2) but n=10^6 may be ok for O(log n)
- Cant directly compare space and time complexities

Application Needs

- Number of items that will be present at a time
- Size of each item
- Memory capacity of machine
- Frequency of inserts and lookups
- How important is low latency for insert & lookup?
- How predictable do you want the latency for operations to be?

Complexity

- Stack.push() as linked list: O(1), insert at head
- BST.search(key): O(log n) expected when balanced, O(n) worst case when skewed