Announcements

- Homework 2, please get started

- Reading:
  - Chapter 4.4, 4.7
  - Skip 4.3.5, 4.5
  - Next time
Question

- So what is the point of a tree structure ??

- What about a BST ??
- One of the nice things about binary trees are some interesting mathematical properties inherit in the tree structure
- Help in analyzing our running times
- We can take advantage of them to further design DS
Understanding trees

- It's clear that the tree expands as it goes lower
  - Exponential growth

- Can you prove that at level $i$ the binary tree has $2^i$ slots??
Proof structure of binary tree

- **Show:**
  - at 0 it has $2^0$ slots

- **Assume**
  - ??

- **Prove**
  - At level $N$ each of the $2^N$ slots will have at most 2 children
  - so next level: $2 \times 2^N = 2^{N+1}$
  - which means level $N+1$ has $2^{N+1}$
Prove

- So now we want to know about where the nodes are living in our tree

- If we have N internal nodes in a binary tree, how many external nodes will we have??
If N internal nodes, we have N+1 external nodes

Idea:
- for the tree to move from N internal to N+1 internal, an external must add children.
- subtract one external (its now internal)
- can have up to +2 children
Definition

- Complete Binary Tree:
  - this is a binary tree which is filled in left to right on each level so no empty spots
  - We try to max the efficiency of our tree
  - So we can store it in an array

- Some implications
- A[i]
  - left is A[i*2]
  - right A[i*2+1]
  - parent A[i/2]
  - root A[1]
    - can save A[0] for other stuff

- Question:
  - any ideas on how to check if A[i] is a leaf ??
Height

- You hopefully remember how to get height of a tree
- For a binary tree with $N$ internal nodes, its height is between $\log n$ and $n-1$
  - worst case
    - linked list case
  - best
    - any ideas ??
To get good use out of the Tree structure, we want to place things as high as possible.

We want it as balanced as possible as a BST to give us best case scenario.

Will need to continually keep the BST balanced while not sacrificing the runtime or BST property.

What is the BST property by the way??
with regular BST

- insert
  - 5, 14, 3, 12

- now:
  - 14, 12, 5, 3
AVL tree

- From the outside an AVL tree looks exactly the same.

- If you look at the code, we only modify insert to do special things and check for specific conditions.
Adelson-Velskii & Landis

- AVL Tree
  - data structure which keeps the trees (almost) balanced at all times, using AVL property

- AVL Property
  1. \( H(\text{right}) == H(\text{left}) \)
  2. \( H(\text{right}) \) differs from \( H(\text{left}) \) by 1

\( H(\text{empty tree}) = -1 \)

\( H(\text{node}) = \) longest path
run time

because the tree is always balanced:

- The runtime are always:
  - inserts = log n
  - find = log n
AVL Rotation

- rotation is the operation of keeping an AVL tree balanced

- Remember that because it's always an AVL tree, the only way to unbalance it is by insert/delete a node

- if \( \alpha \) is the node that needs rebalancing, it means \( \alpha \) subtree was edited in some way

- divides itself into 4 cases
1. Just inserted into left subtree of left child
2. Just inserted into right subtree of left child
3. Just inserted into left subtree of right child
4. Just inserted into right subtree of right child

- single rotations 1,4
- double rotations 2,3

- lets show this graphically:
- insert 5,2,8
- 1,4,7,3

- now insert 6
  - which is unbalanced ?
  - which rule is used ?

- how do you code this ??
code

- Avl insert
code

- Avl rotate single
code

- Avl double rotate
question

- so using an AVL tree what do we get if we print in-order traversal ??
- so using the AVL tree to sort, what is the cost of sorting n items ??

- which tree operations are involved ??
Limitations

- One limitation is that with huge tree structure might not fit into memory.

- As you need to travel down the tree, you take a performance hit at every level down.

- One solution: store more information along the path.
B Tree

- great when can’t fit the data in memory
- type of search tree
- trying to make getting to data fast
  - increasing width will make getting to data faster

- \( M \)
  - maximum number of children per internal node (we did 3)

- \( L \)
  - maximum number of data items per leaf
B-trees

- 2-3 B-trees are tree DS found in databases and file systems
  - automatically balanced
  - keep all data at leaf level
  - non leaf keys guide search by storing 1 or 2 keys (2,3) children
  - root is either leaf or between 2,3 children
Rules

1. All data stored at leaf level
2. non leaf nodes store M-1 keys
3. root is either leaf or has between 2 and M children
4. All internal nodes have between M/2 and M children
   1. restricts branching factor
5. All leaves are on the same depth and have between L/2 and L children
Key
- the internal node key represents the smallest value on the \(i+1\)th subtree

Leafs can be any DS you choose

Any ideas on the advantage of the leaf system here??
Example

- simple case: room in leaf
  - insert: 3, 1, 5
- Harder: split when full
  - inserting: 8, 7, 6
- Even Harder: move up the tree
Data Structure

- we spoke about queues, but many times would like to add a priority to the Queue DS

- Priority Queue
  - Insert
  - FindMin

- Example: phone call service center
Implementations

- Any ideas?
Implementations

- sorted linked list
  - insert – $O(???)$
  - findmin – $O(???)$

- unsorted linked list
  - insert
  - findmin
Question

- What about using a BST??
  - insert
  - findmin
Heap

- A heap is an implementation of a priority queue
  - property: it is a binary tree that is completely filled except on the bottom level

- Any advantages ??
Heap

- implementation of priority queue
- Heap order property:
  - any parent is as small or smaller than its children

- can use array representation:
  - no links to manage
  - but need to estimate largest size ahead of time
used everywhere

- service priority
- Operating systems – juggling threads, processes and processors
Example

- When we are interested in $k^{th}$ smallest number in a large number set

- can sort and then pick it out
  - what is the run time??

- can create BST
  - how will this help
  - what is the run time?

- can create heap and do $k$ findmin
  - what is the run time??
percolate up

- insert operation
- idea: bubble up till we reach correct location on the tree

1. create hole in tree to hold value
2. Does it fit ....done
3. else: switch with parent and try again

Runtime for this ??
so now we need to find min....

what is the run time for finding the min ??

what if we want to find and delete ??
percolate down

- pull out the root
- put hole
- swap with smaller child
- bubble down the hole
run time

- how long would it take me to find an item of specific priority??

- any ideas on how to help this?
building a heap

- given a set of N items what is the fastest way of building the heap ??
easy solution

- just do N inserts
  - worst case will be $O(n \log n)$

- when heaps invented that was best way

- they actually have a linear time algorithm....any ideas ?
linear time build

- start in middle
- work way back up

- why it works?
D-heaps

- we were doing binary heaps, but no reason can't have larger branching factor

- issues:
  - when would be the best time to use heaps?
  - how to structure it if can't fit entire heap in memory?
change of pace

- so can we summarize the runtimes for the DS we have covered??
Question

- if we have 10,000 items
  - how would you store it to quickly support find?

- now what if you only had 20 items
  - how would this be different
Compression

- Many times we need to compress information
  - scaling factor
  - resource allocation
  - over promise

- lossy compression
  - JPEG
  - PNG

- lossless compression
  - when would this be important?
  - TIF
  - BMP
  - .zip
is everyone familiar with the tar program??
  - usage
  - how it works ?
- ASCII encodes each character as a 7 bit value
- the idea of compression, is to find a better way of representing your information
  - idea: instead of using uniform length codes for everything, use less bits for higher occurring information parts

- Huffman trees allow you to create very good lossless compression tables to be able to quickly compress text