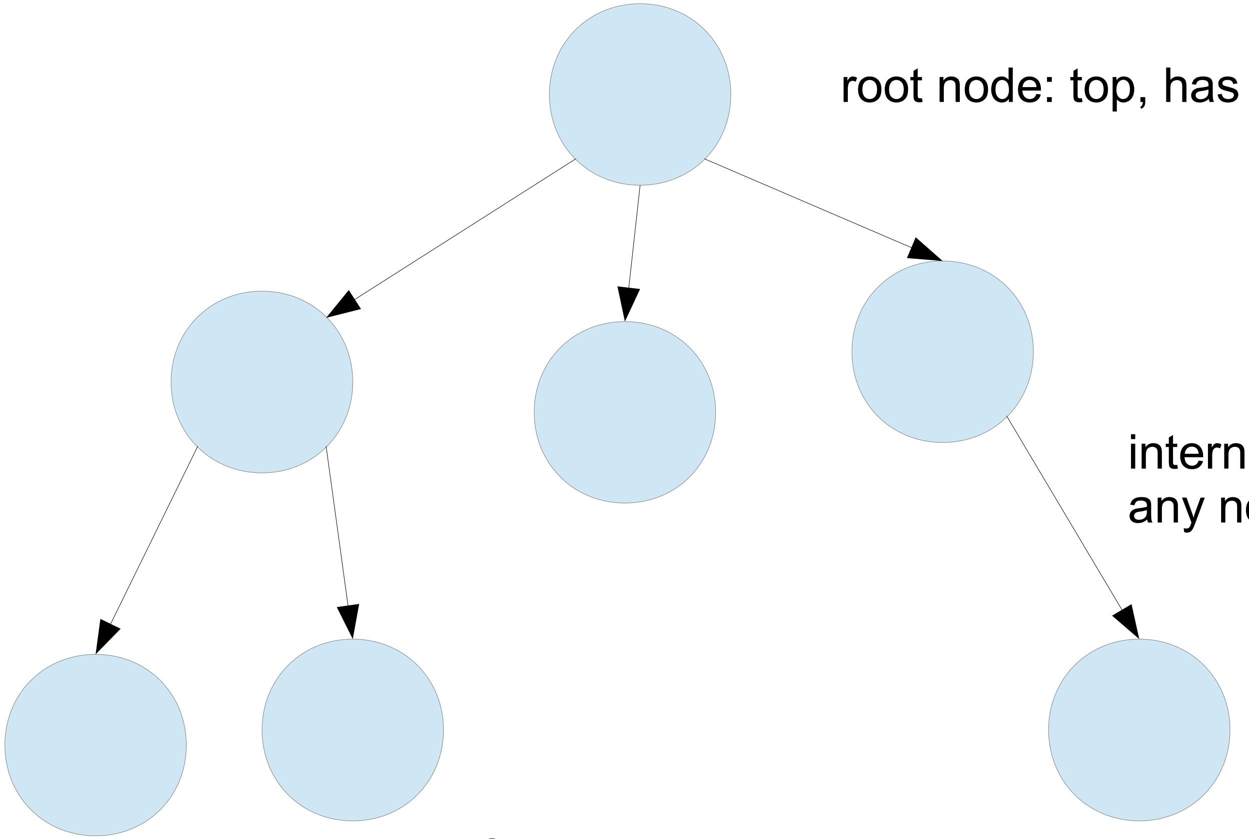
Trees, search trees

- linked lists are nice simple dynamic data structure, but are inefficient for anything but sequential access or front/back access
- many other data structures are used for scenarios where you need dynamic sizing but non-sequential access
- trees: each internal node has pointers to zero or more “child” nodes, rather than left/right neighbours
- topmost node is called the root (has no “parent”)
- gives a heirarchical structure, can have a large number of nodes yet each can have a short path from the root

tree: root, leaf, internal nodes



root node: top, has no parents

internal nodes:
any node with both a parent and children

leaf nodes, any node with no children

typical tree operations

- initialize a new, empty tree
- insert a new node into the tree (but where?)
- search the tree for node containing specific data
- print the tree contents (but in what order?)
- remove nodes from the tree (how to restructure?)
- delete all the nodes in a tree

Implementation

- can be much like a linked list
- each node can be a struct or class, with data fields and pointer fields (but for parent/children instead of next/prev)
- tree itself maintains a pointer to root node instead of front/back

k-ary trees

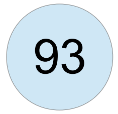
- binary trees: each node has (at most) two children
  - ternary trees: each node has (at most) three children
  - quad trees, etc
- binary search trees widely used
  - organized to be rapidly searchable on some data field
  - small values go down one side of tree, large values the other
- balanced binary search trees
  - binary search treesm but re-organized as/when needed to be as dense as possible (use fewest levels possible, try to keep every node as close to the root as possible)

binary search trees

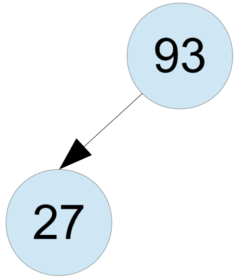
- the backbone of many common searchable data structures (well, balanced binary search trees)
- assumes each node contains some “key” value we're using as the basis for searches
- first node created/inserted into a tree becomes the root
- subsequently, any value inserted goes into left/right subtrees based on whether its key is smaller or larger
  - applied recursively as you go down the tree
- when inserting or searching you can always look at current node's key to see which direction to go

example: inserting values

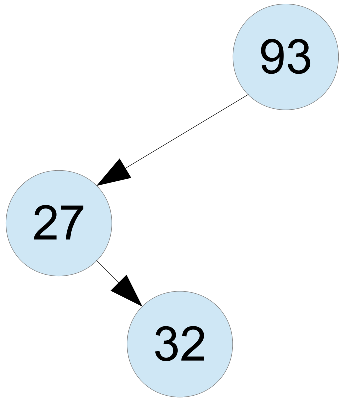
- suppose our keys are integers, first one inserted is a 93



- then someone inserts 27: smaller than 93 so goes down left

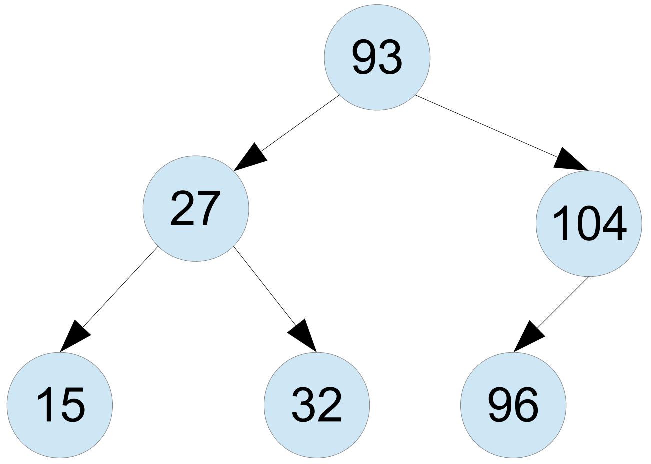


- then someone inserts 32: smaller than 93 so goes into left subtree of 93, then bigger than 27 so goes into right subtree of 27



example: continued

- suppose then insert 104, then 15, then 96



104 > 93 so goes in right

15 <= 93, then < 27

96 > 93, then <= 104

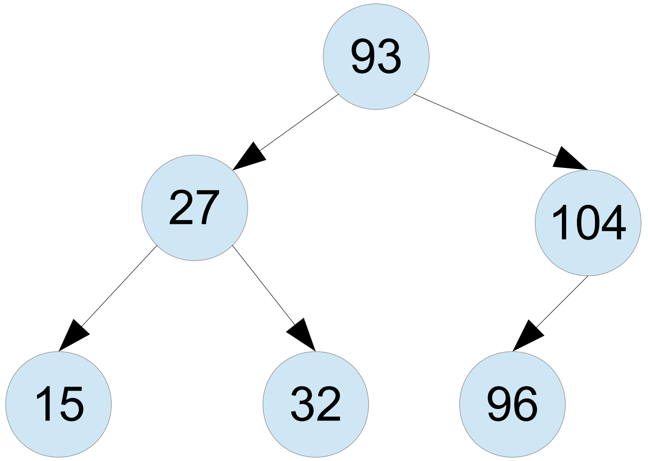
- keep going down left/right using <=, > rule
- insert when you find an empty spot

searching

- when searching, you start at the root
- compare value you're looking for to the current node's key
- go left if search value < node's key
- go right if search value > node's key
- continuing searching down the tree until you find your target value or hit a leaf without finding it

search example

- suppose we were searching for 32



32 < 93 so into its left subtree

32 > 27 so go into its right subtree

found!

- suppose we were searching for 29
  - search would left from 93 then right from 27
  - would try to go left from 32, but nothing there, so value not found

printing

- suppose we want to print the tree content in sorted order (based on the nodes' keys)
- for each node, the smaller values are in its left subtree so want to print those first, then the current node, then the values in its right subtree
- leads to simple recursive algorithm given ptr to a node

```
print(node *n)
   if n isn't null:
      print(n->left)
      cout n's data fields
      print(n->right)

// top-level call would be to print(root)
```

tree traversals

- our print algorithm did left subtree, current node, right subtree
- called an inorder traversal (current node done in middle)
- preorder traversals do current node, then left subtree, then right subtree (called a topological sort, visits parent before subtrees)
- postorder traversals do left subtree, then right subtree, then current node
- the pre/in/post indicates when position the current node is done
- pre/postorder traversals useful for generating prefix/postfix expressions (a bracket-free way to unambigously represent numeric expressions)

sidebar: prefix, infix, postfix expressions

- given expression (x * y) - (a / (b + c))
- tree rep shown on right, leaf nodes contain data, ops in internal nodes
- simple inorder traversal gives infix expression x * y - a / b + c
- simple preorder traversal gives prefix expression - * x y / a + b c
- simple postorder traversal gives postfix expression x y * a b c + / -
- the infix is ambigous without brackets, the infix/prefix are not, wind up being simpler to evaluate