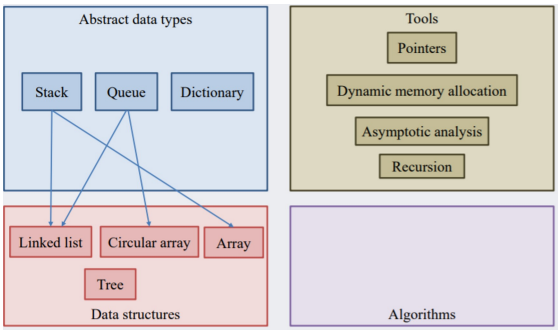
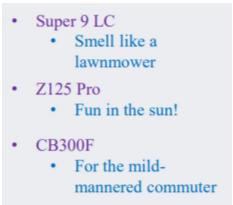
Trees

October 30, 2019 8:02 AM



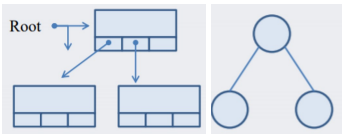
Dictionary ADT

- Abstract data type ADT
- stores pairs of strings, words, definition
- Operations
  - Insert
  - Remove
  - Lookup



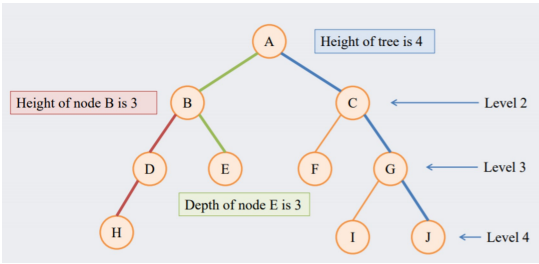
Trees

- linked lists constructed of nodes, consisting of:
  - a data element
  - a pointer to another node
- Trees constructed of nodes, have pointers to one or more nodes
- set of nodes with single starting point = root of tree = root node
- connected graph, there is a path to every node in the tree, a tree has one less edge than the number of nodes




```
typedef struct TreeNode {
        int data;
        struct TreeNode* c1;
        struct TreeNode* c1;
        struct TreeNode* c1;
} TreeNode;
```

- **Tree relationships**
  - node v is child of u, and u and parent of v if there is an edge between and u is above v in tree
  - ex. descendants, ancestors, siblings, cousins
- **Terminology**
  - leaf = node with no children
  - path = sequence of nodes v1...vn where vi is a parent of v_{i+1}
  - subtree = any node in the tree along with all its decendants
  - degree of a node = number of children node has
  - Branching factor = maximum degree of any node in tree
  - binary tree = at most 2 children/nodes, branching factor = 2



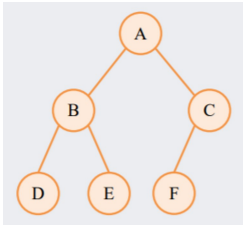

```
typedef struct BinaryNode {
        int data;
        struct BinaryNode* left;
        struct BinaryNode* right;
} BinaryNode;
```

Measuring trees

- height of node v = number of nodes in the longest path from v to a leaf
- tree height = root height, empty tree height = 0, tree root height = 1
- depth of a node v = number of nodes path from v to the root
  - referred to as the level of a node

Perfect Binary Trees

- binary tree is perfect if
  - no node has only one child
  - all leaves have the same depth
- perfect binary tree height h has 2^h-1 nodes, of which 2^{h-1} are leaves
- **Height**
  - each level doubles number of nodes
    - Level 1 = 1 node 2^0
    - Level 2 = 2 nodes 2^1
    - Level 3 = 4 nodes 2^2



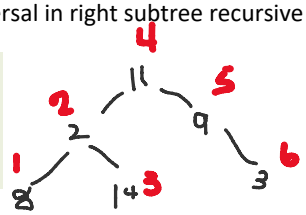
Complete: Tree is complete if leaves are on at most 2 different levels, bottom level is completely filled, and leaves on bottom are as far left as possible

Binary tree traversal

- visits each node in the tree, and do some work
- naturally recursive
- traverse left subtree, then traverse right subtree
- 3 traversal methods:
  - **inOrder**
    - traversal in left subtree recursive call, then traversal in right subtree recursive call

```
void inOrder(BNode* nd)
{
  if (nd != NULL)
  {
    inOrder(nd->left);
    visit(nd);
    inOrder(nd->right);
  }
}
```

```
typedef struct BNode
{
  int data;
  struct BNode* left;
  struct BNode* right;
} BNode;
```



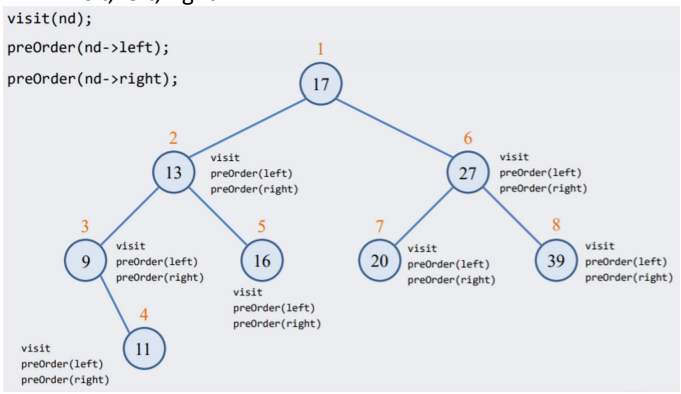
    - output: 8,2,14,11,9,3
    - initializing

inOrder(nd->left);
visit(nd);
inOrder(nd->right);

  - **preOrder**

visit(nd);
preOrder(nd->left);
preOrder(nd->right);

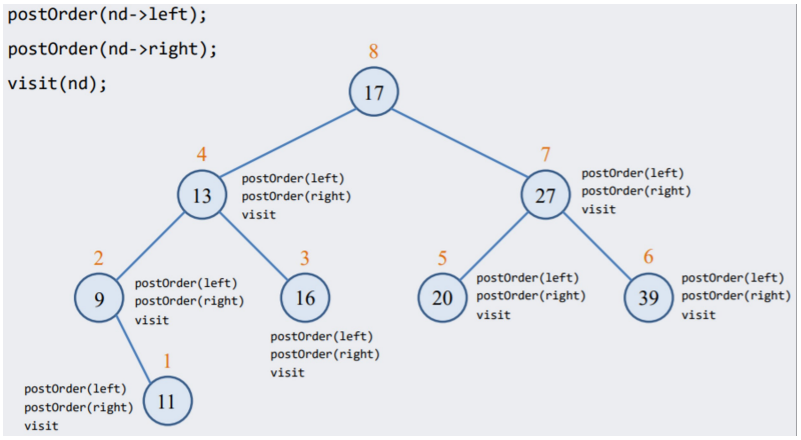
    - visit first, top downwards, left bias. once finish all steps, go back to previous
    - visit, left, right



  - **postOrder**

postOrder(nd->left)
postOrder(nd->right);
visit(nd);

    - start with recursive calls first
    - bottom to upwards
    - left, right, visit

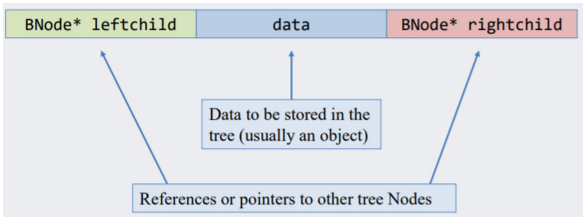


- **Level-order traversal**
  - visit every node in level before working on next level
  - use Queue

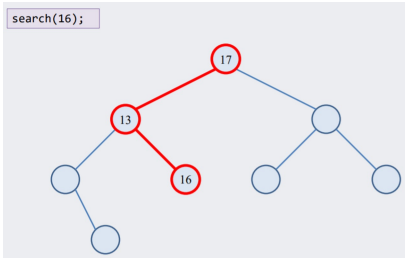
Binary Search trees

- binary tree with special property, for all nodes in tree:
  - nodes in left subtree have labels less than label of subtree's root
  - nodes in right subtree have labels greater than or equal to label of subtree's root
  - fully ordered

  inOrder(nd->leftchild);
  visit(nd);
  inOrder(nd->rightchild);

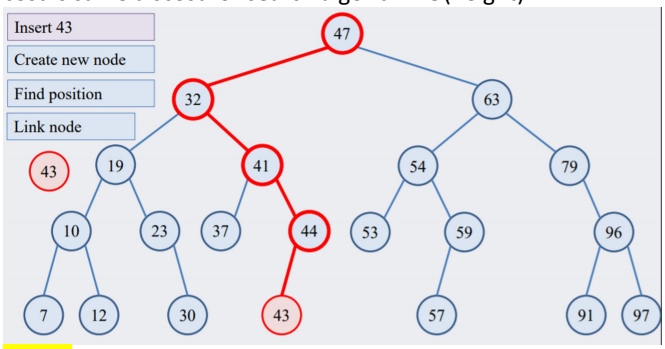


- can be implemented using a reference structure
- Tree nodes contain data and two pointers to nodes
- To find a value in BST, search from root node:
  - if target < value in node, search left subtree
  - if traget > value in node, search right subtree
  - else return true (or a pointer to data etc.)
- 1 comparison for each node on path or worst case = height of tree
- **Search:** can be implemeted iteratively or recursibely




```
int search(BNode* nd, int key) {
  if (nd == NULL) return FALSE;
  else if (nd->data == key) return TRUE
  else {
    if (key < nd->data)
      return search(nd->left, key);
    else
      return search(nd->right, key);
  }
}
```

- BST property must hold after insertion
  - new node must be inserted in correct position
  - position found by performing a search
  - if search ends at null left child of node, make its left child refer to new node
  - if search ends at right child of a node, make its right child refer to the new node
- cost is same a scost for search algorithm O(height)

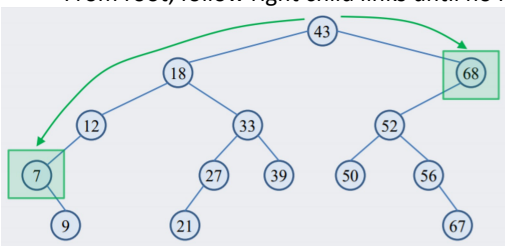


- **Insert:** Can be implemented iteratively or recursively

```
BNode* insert(BNode* nd, int key) {
  if (nd == NULL) {
    BNode* newnode = (BNode*) malloc(sizeof(BNode));
    newnode->data = key;
    newnode->left = NULL;
    newnode->right = NULL;
    return newnode;
  }
  else {
    if (key < nd->data)
      nd->left = insert(nd->left, key);
    else
      nd->right = insert(nd->right, key);
    return nd;
  }
}
```

Find Min, Max

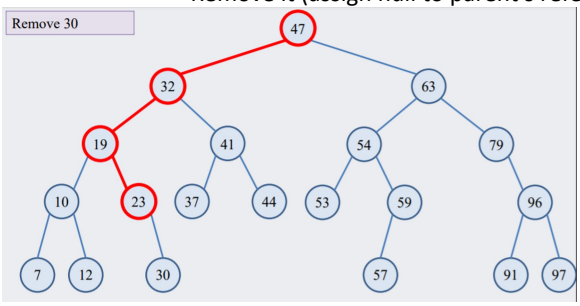
- **Find Min:**
  - from root, keep following left child links until no more left child exists (NULL)
- **Find Max:**
  - From root, follow right child links until no more right child exists



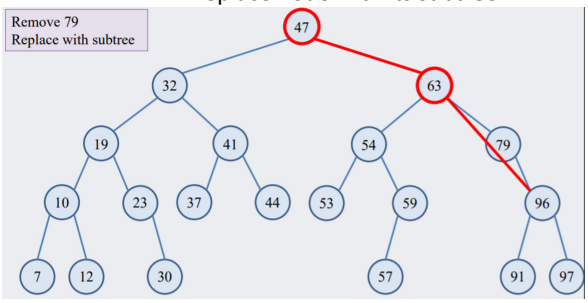

```
int findMin(BNode* nd) {
  BNode* curr = nd;
  if (nd == NULL)
    return -1;
  else {
    while (curr->left != NULL)
      curr = curr->left;
    return curr->data;
  }
}
```

BST Removal

- Cases:
  - **Node to be removed has no children**
    - Remove it (assign null to parent's reference)



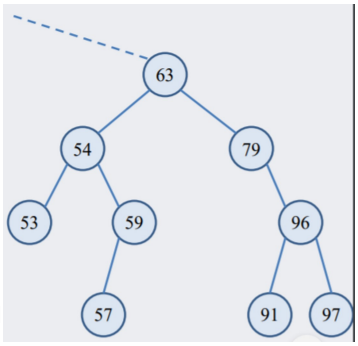
  - **Node to be removed has one child**
    - Replace node with its subtree



  - **Node to be removed has two children**
    - Look at next node, at both children
    - ex. removing a node with one child
      - ☐ Need to find the node to remove and its parent
      - ☐ Need to know if node to be removed is left or right child
      - ☐ **Left or right?**

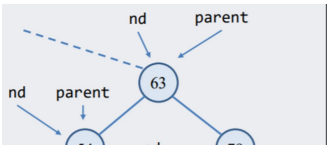
```
while (nd != target) {
  if (nd == NULL)
    return;
  if (target < nd->data) {
    parent = nd;
    nd = nd->left;
    isLeftChild = true;
  }
  else {
    parent = nd;
    nd = nd->right;
    isLeftChild = false;
  }
}
```

November 04, 2019

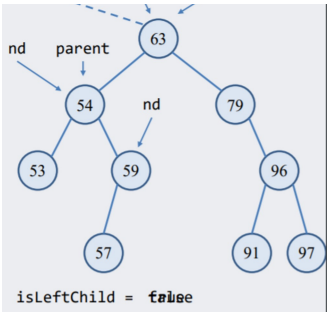


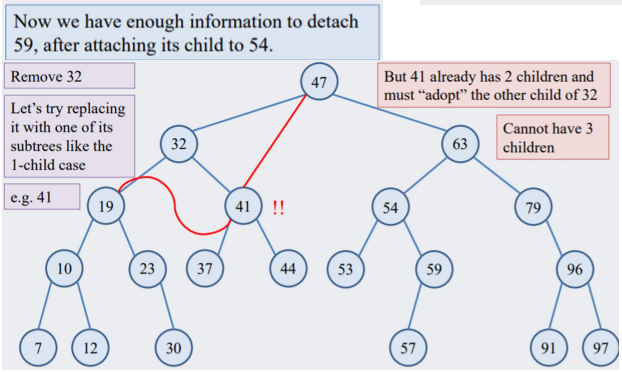
Remove 59

```
while (nd != target) {
  if (nd == NULL)
    return;
  if (target < nd->data) {
    parent = nd;
```

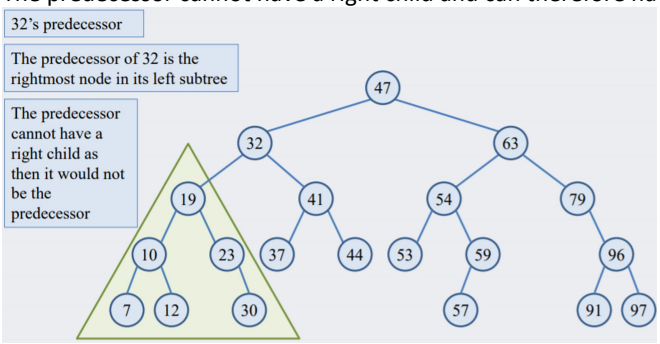


```
while (nd != target) {
  if (nd == NULL)
    return;
  if (target < nd->data) {
    parent = nd;
    nd = nd->left;
    isLeftChild = true;
  }
  else {
    parent = nd;
    nd = nd->right;
    isLeftChild = false;
  }
}
```

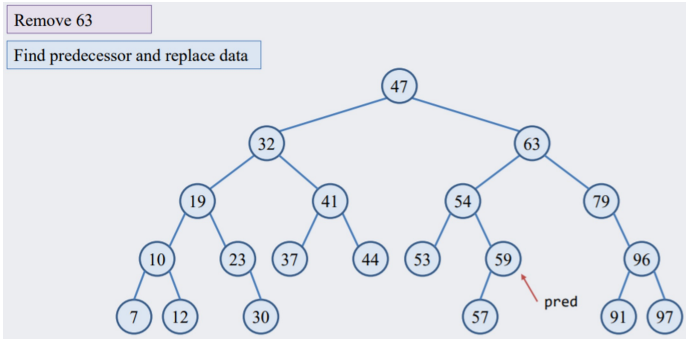


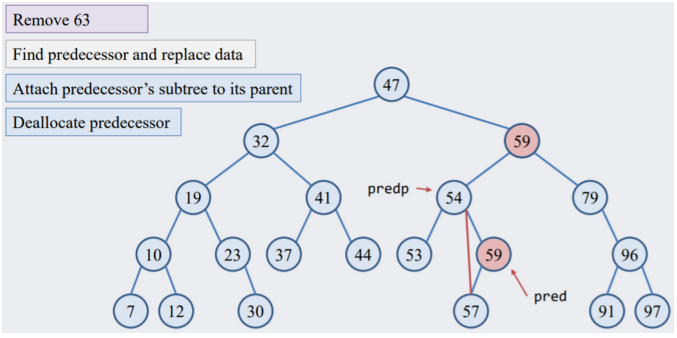


- Instead of replacing node with 2 children, find its predecessor
  - Right most node of its left subtree
  - predecessor is the node in the tree with the largest value less than the node's value
- The predecessor cannot have a right child and can therefore have at most one child



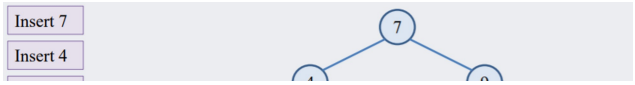
- **Predecessor**
  - must be the largest value less than its ancestor's value
    - it is to the right of all of the nodes in its ancestor's left subtree, must be greater than them
    - it is less than the nodes in its ancestor's right subtree
    - can have at most one child
  - **Successor**
    - smallest value greater than its ancestor's value
    - cannot have a left child
    - can be used to replace a removed node





- efficiency of BST operations depends on height of tree
- all 3 operations search insert and delete are O(height)
- if tree is complete, the height is log(height)

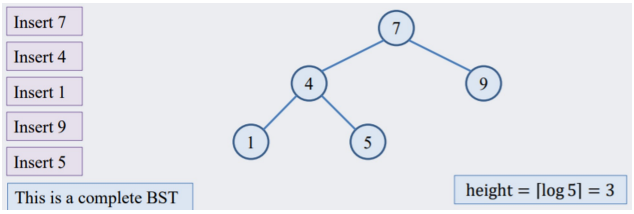
Height of a BST



- height = [log(nodes)]

- If tree is complete, the height is log(height)

Height of a BST
- height = [log(nodes)]



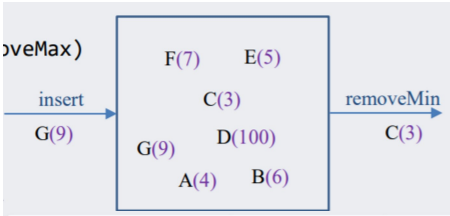
BST efficiency
- depends on height of tree
- Dictionary ADT:

	Insertion	Removal	Search
Unordered doubly linked list	O(1)	O(n)	O(n)
Ordered Array	O(n)	O(n)	O(logn)
BST	O(n)/O(logn)	O(n)/O(logn)	O(n)/O(logn)

Priority Queues (dictionary)
- ex. TODO list, find task with highest priority

7 – Grade homework and labs
2 – Vacuum home
8 – Renew driver's license
1 – Sleep
9 – Fix overflowing sink
2 – Eat
8 – Lecture prep
1 – Bathe

- Collection organised, allow fast access to and removal of the largest/smallest element
  - prioritisation is a weaker condition than ordering
  - order of insertion is irrelevant
  - element with higest priority removed first
- **Operations**
  - create
  - destroy
  - insert
  - removeMin (removeMax)
  - isEmpty
- **Binary heap**
  - binary tree with properties:
    - heaps complete, all levels except bottom filled in
    - leaves on bottom as far left as possible
  - partially ordered
    - for max heap, value of node is at least as large as childrens value
    - for min heap, value of node is no greater than its children's values



Structure	insert	removeMin
Unordered list	$O(1)$	$O(n)$
Ordered list	$O(n)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$

A balanced BST can do insert and removeMin in $O(\log n)$, but we cannot guarantee balance

Binary heap	$O(\log n)$	$O(\log n)$