Binary Search Tree

- *Recursion Revisited*
- *binary search tree Implementation*
    - *traversal - inorder, preorder, postorder, levelorder*
    - *minimum, maximum,*
    - *predecessor, successor*
    - *height*
    - *clear*
    - *contains*
    - *grow*
    - *trim*

bunnyEars(): counting bunny ears in recursion

```
// each bunny has two ears.
int bunnyEars(int bunnies) {
  if (bunnies == 0) return 0;
  return 2 + bunnyEars(bunnies-1);
}
```

funnyEars(): counting funny ears in recursion

```
// even numbered funny has two ears, odd numbered funny three.
int funnyEars(int funnies) {
  if (bunnies == 0) return 0;

  if (funnies % 2 == 0)
    return 2 + funnyEars(funnies - 1);
  else
    return 3 + funnyEars(funnies - 1);
}
```

Binary search trees

size(): in doubly linked list

```
int size(pList p) {
  int count = 0;
  for (pNode c = begin(p); c != end(p); c = c->next)
    count++;
  return count;
}
```

size(): in singly linked list

```
int size(pNode node) {
  if (node->next == nullptr) return 0;
  return 1 + size(node->next);
}
```

Binary search trees

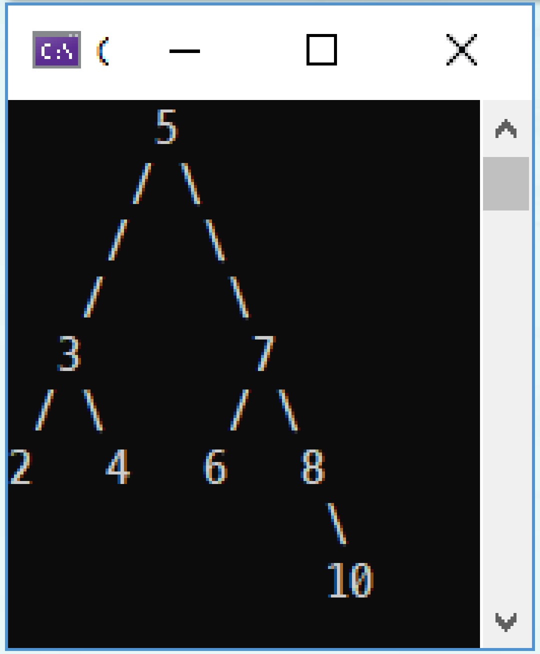
size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  return 
}
```

Binary search trees

size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  cout << " size at: " << node->key << endl;
  return 1 + size(node->left) + size(node->right);
}
```



Binary search trees

size: Count the number of nodes in the binary tree recursively.

```
int size(tree node) {
  if (node == nullptr) return 0;
  return 1 + size(node->left) + size(node->right);
}
```

height: compute the max height(or depth) of a tree.
// It is the number of nodes along the longest path from the root node
// down to the farthest leaf node.

```
int height(tree node) {


}
```

Binary search trees

BST Node structure:

Key	
Left	Right

```
struct TreeNode{
  int        key;      // sorted by key
  TreeNode* left;     // left child
  TreeNode* right;    // right child
  TreeNode(const int k = 0, Tree* l = nullptr, Tree* r = nullptr) {
    key = k; left = l; right =r;
  }
  ~TreeNode() {}
};
using tree = TreeNode*;
```

grow: insert a new node with given key in BST

```
tree grow(tree node, int key) {
  if (node == nullptr) 
  if (key < node->key)   // recur down the tree
    grow(node->left, key);
  else 
    grow(node->right, key);
  else  
    cout << "grow: the same key " << key << " is ignored.\n";

  return node;
}
```

Binary search trees

grow: insert a new node with given key in BST

```
tree grow(tree node, int key) {
  if (node == nullptr) return new Tree(key);
  if (key < node->key)  // recur down the tree
    grow(node->left, key);
  else if (key > node->key)
    grow(node->right, key);
  else
    ;

  return node;
}
```

Binary search trees

inorder traversal: do inorder traversal of BST.

```
void inorder(tree node) {
    if (node == nullptr) return;

    inorder(node->left);
    cout << node->key;
    inorder(node->right);
}
```

```
void inorder(tree node, vector<int>& vec) {
  if (node == nullptr) return;

  inorder(node->left, vec);

  inorder(node->right, vec);
}
```

inorder traversal: do inorder traversal of BST.

```
void inorder(tree node) {
    if (node == nullptr) return;

    inorder(node->left);
    cout << node->key;
    inorder(node->right);
}
```

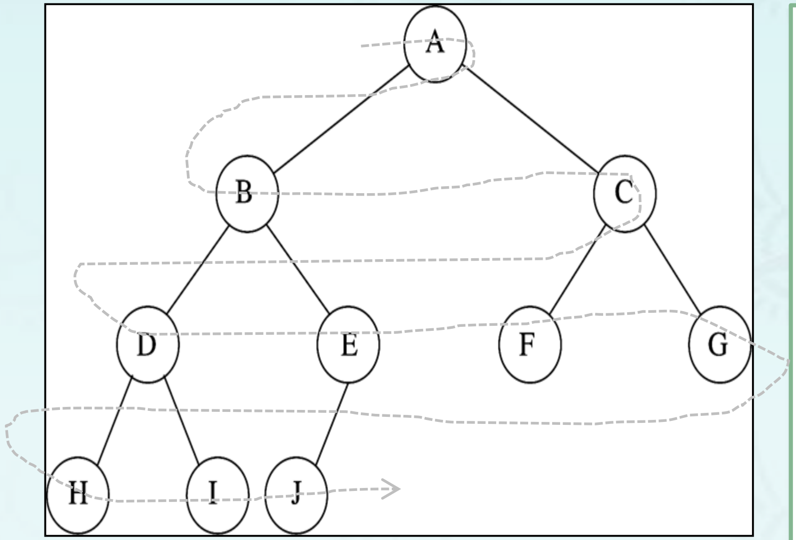
```
void inorder(tree node, vector<int>& vec) {
  if (node == nullptr) return;

  inorder(node->left, vec);
          _
  inorder(node->right, vec);
}
```

```
case 'l':
  cout << "\n\tinorder:    ";
  vec.clear();
  inorder(root, vec);
  for (int i : vec)
    cout << i << " ";
```

Binary search trees

1. **Depth first** search(DFS) – preorder, inorder, postorder traversal
2. **Breadth first** search(BFS) - **level-order** traversal



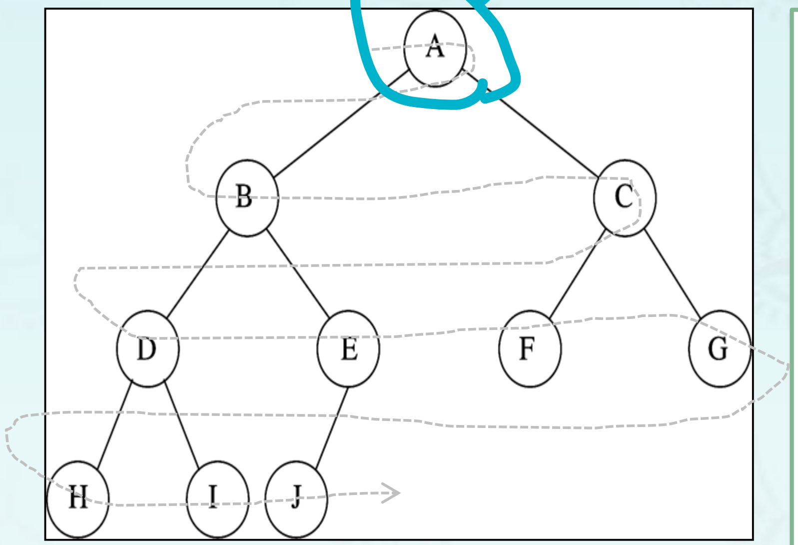

```
#include <queue>
#include <vector>

void levelorder(tree root, vector<int>& vec)
•  Visit the root. if it is not null, enqueue it.
•  while queue is not empty
    1. que.front() – get the node in the queue
    2. save the key in vec.
    3. if its left child is not null, enqueue it.
    4. if its right child is not null, enqueue it.
    5. que.pop() – remove the node in the queue.
```

Binary search trees

1. **Depth first** search(DFS) – preorder, inorder, postorder traversal
2. **Breadth first** search(BFS) - **level-order** traversal

que




```
#include <queue>
#include <vector>

void levelorder(tree root, vector<int>& vec) {
  queue<tree> que;
  if (!root) return;
  que.push(root);
  while ...{


     cout << "your code here\n";


  }
}
```

minimum, maximum: returns the node with min or max key.
Note that the entire tree does not need to be searched.

```
tree minimum(tree node) {   // returns left-most node key


}
```

```
tree maximum(tree node) {   // returns right-most node key


}
```

predecessor, successor:

```
Input: root node, key
output: predecessor node, successor node

1.If root is nullptr, then return

2. if key is found then
    a. If its left subtree is not nullptr
        Then predecessor will be the right most
        child of left subtree or left child itself.
    b. If its right subtree is not nullptr
        The successor will be the left most child
        of right subtree or right child itself.
    return
```

Binary search trees

trim:** remove node with the key and return the new root.

```
tree trim(tree root, int key) {
  if (root == nullptr) return root;// base case
  if (key < root->key)
    root->left = remove(root->left, key);
  else if (key > root->key) {
    root->right = remove(root->right, key);
  else if (root->left && root->right) {
    cout << "your code here: node with two children\n”;
  }
  else {
    cout << “your code here: node with one child or no child)\n”;
  }
  return root;
}
```


Prof. Youngsup Kim, idebtor@gmail.com, Data Structures, CSEE Dept., Handong Global University

Binary search trees

http://visualgo.net/bst