SULEYMAN SULEYMANZADE

NON LINEAR DATA STRUCTURES: BINARY TREES

ALGORITHMS AND DATA STRUCTURES

BINARY TREES

Binary trees

- Non linear DS
  - Trees
- Binary tree
  - Overview
  - Structure
  - Implementation
- Binary Search tree
  - Overview
  - Structure
  - Implementation

NON LINEAR STRUCTURE

AVL Trees
Binary Trees
BST
T-tree
Heap
Fenwick tree
Red-Black trees
Treap
Huffman tree
Radix tree
Segment tree
AST

Trees

Directed
Undirected
Labeled
Unlabeled
Cyclic
Acyclic
Sparse
Dense
Weighted
Unweighted
Simple
Non-Simple
Embedded
Topological

Non-Linear data structures

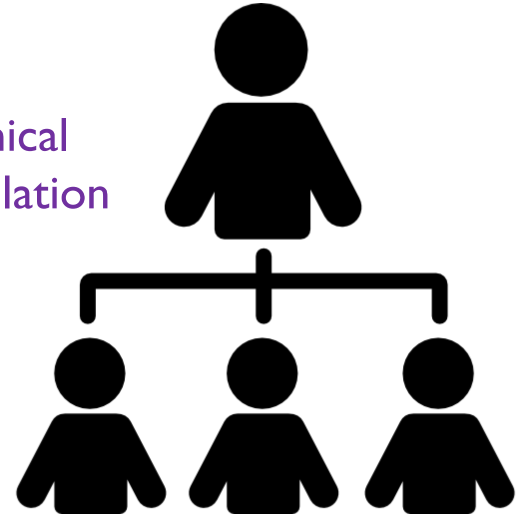
Graphs & Flows

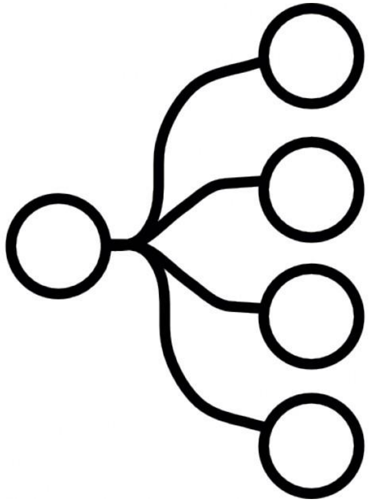
BINARY TREE USAGE

Applications

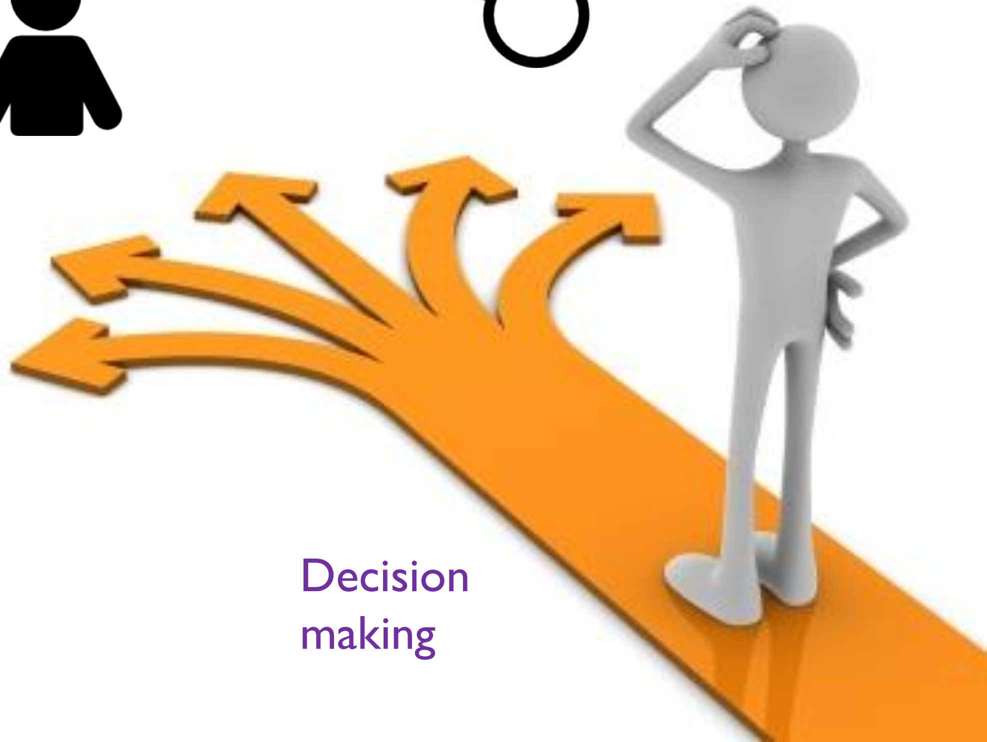
- Hierarchical data manipulation
- Easy to search information
- Manipulation of sorted lists of data
- Router algorithms
- Workflow for compositing digital image for visual effects
- Forms of multi-stage decision-making tree

Hierarchical
Data Manipulation





Router
Algorithms



Decision
making

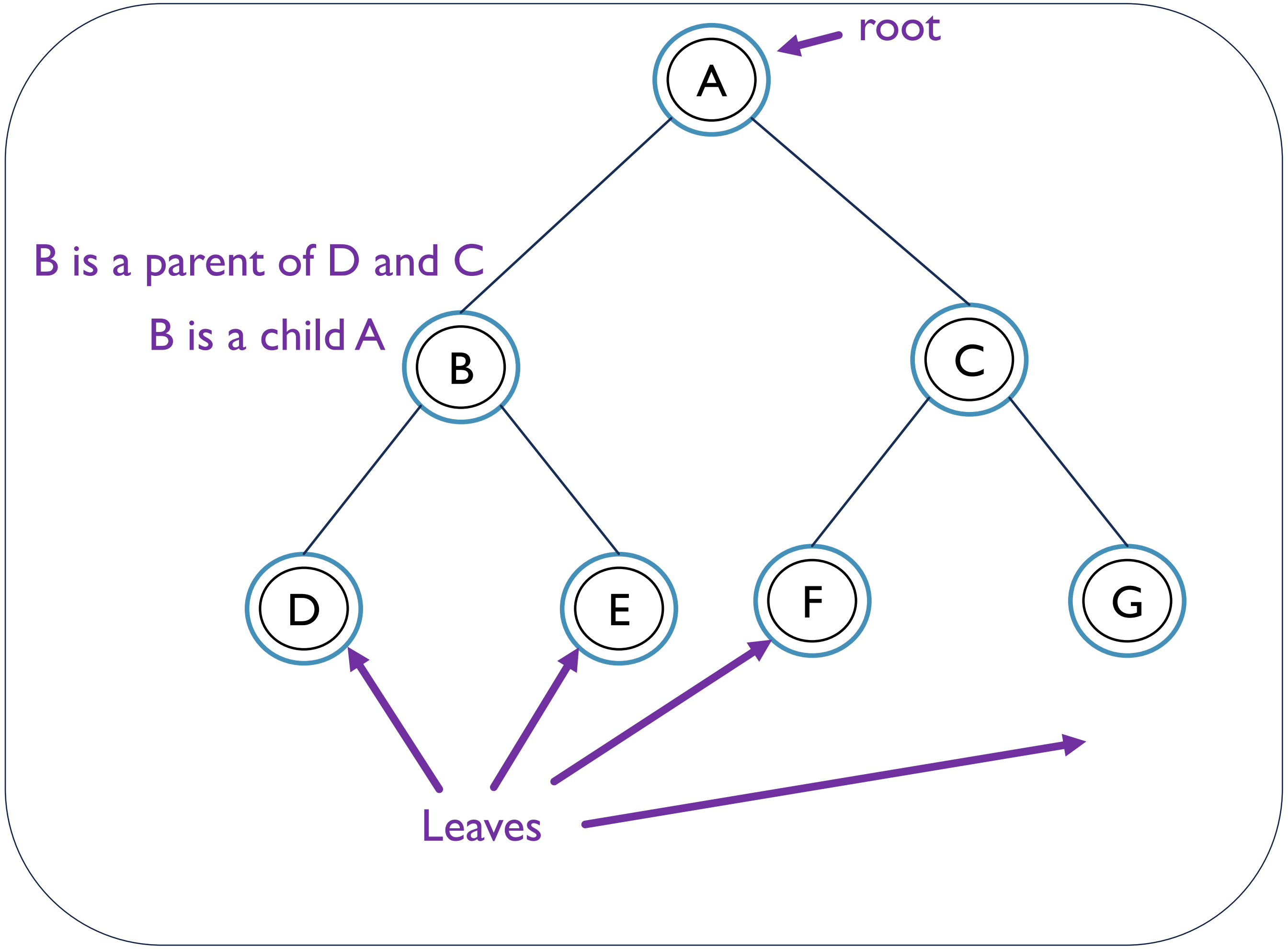
BINARY TREE

STRUCTURE AND ALGORITHMS

BINARY TREE STRUCTURE

Binary tree

- Non Linear data structure
  - Unlike arrays, Lists, Stacks, queues- trees are hierarchical data structures
- Elements that are directly under an element are called children
  - B is a children of A
  - D is a children of B
- Elements that are upon of an elements are called parent
  - C is a parent of F
  - A is a parent of B and C



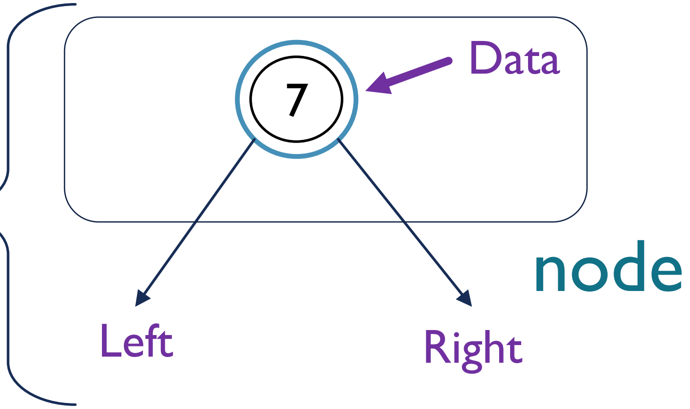
STRUCTURE

Representation

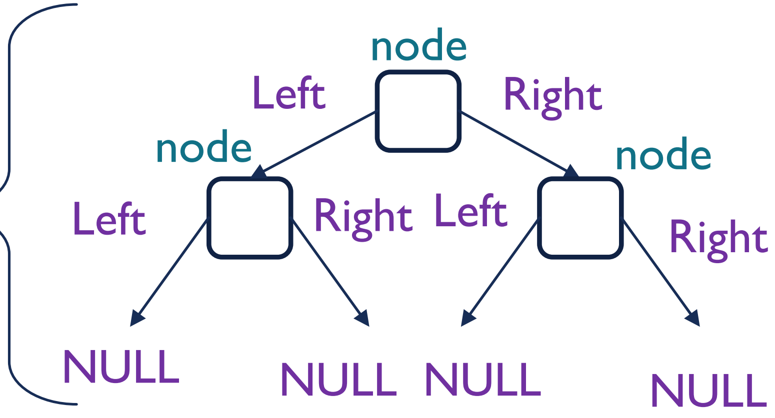
- A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.
- Elements
  - Data
  - Pointer to the left child
  - Pointer to right child

```
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

Building block



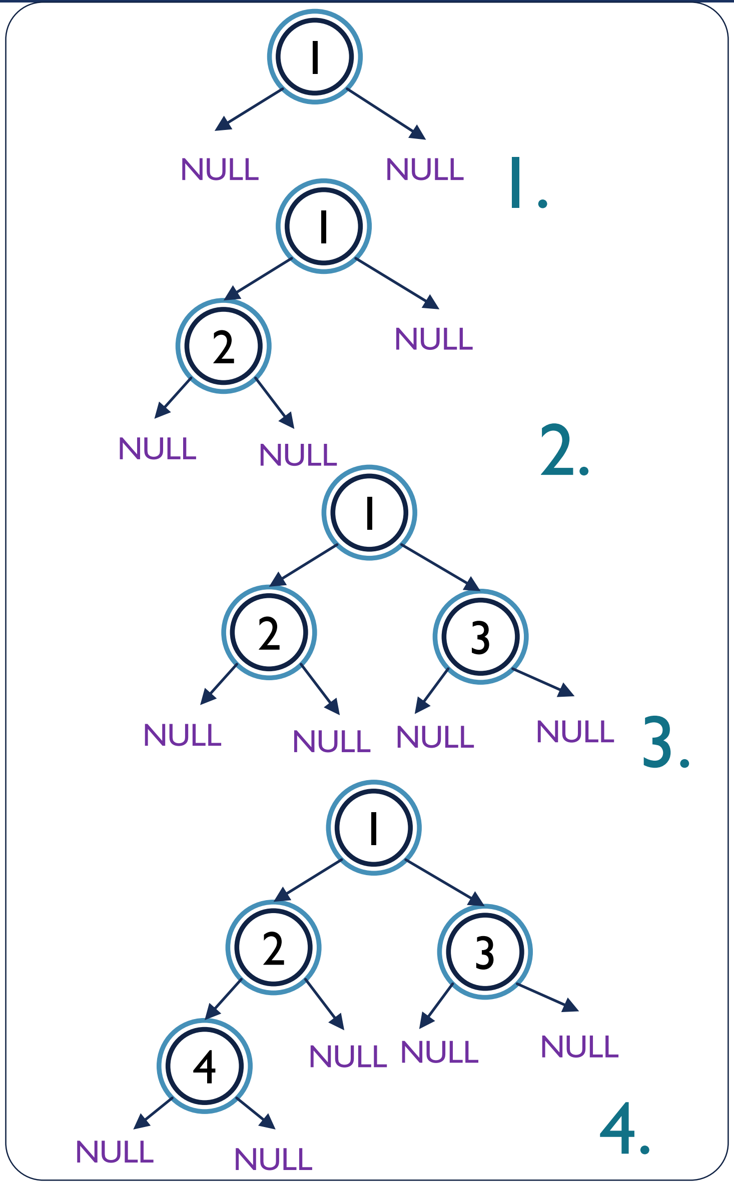
Tree



ADD THE NODE TO THE STRUCTURE

```
#include<iostream>
using namespace std;

struct node{
  int data;
  struct node *left;
  struct node *right;
};

node* newNode(int data) {

  node* nn = new node();

  nn->data = data;
  nn->left = NULL;
  nn->right = NULL;

  return nn;
}
```

```
int main(){
  struct node *root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  system("pause");
  return 0;
}
```



BINARY SEARCH TREES

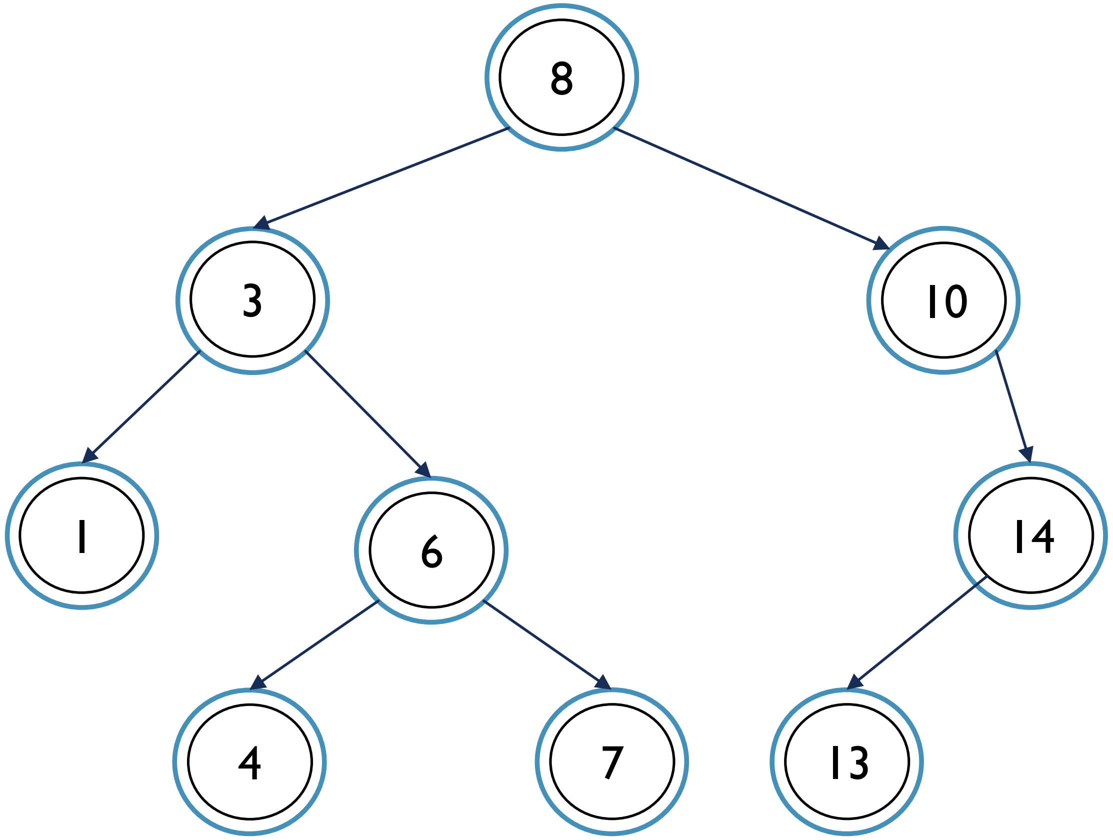
STRUCTURE AND ALGORITHMS

BINARY SEARCH TREE (BST)

Properties

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes



Searching

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

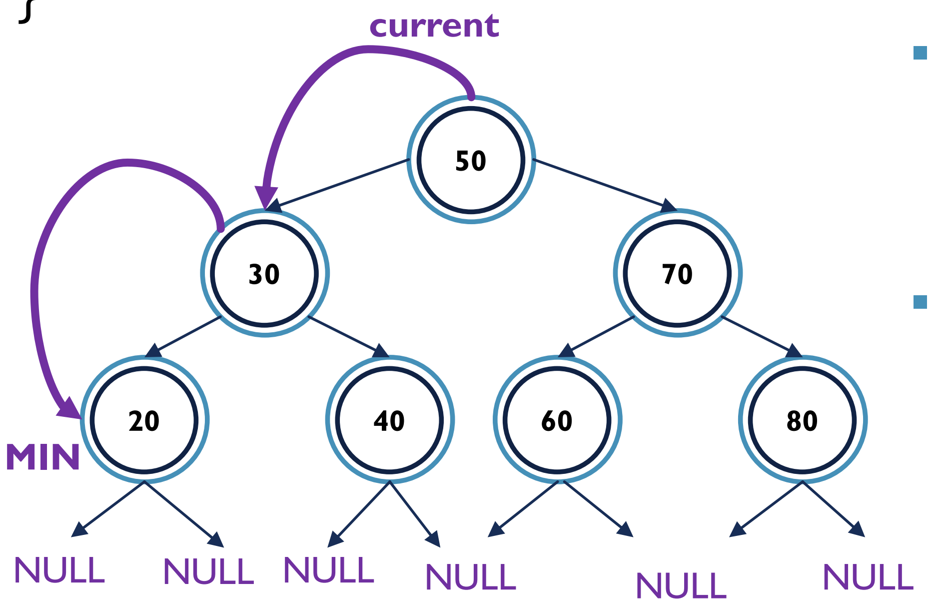
INSERTING INTO THE BST

```
#include<iostream>
using namespace std;

struct node
{
  int key;
  node *left, *right;
};

node *newNode(int item){
  node *temp = new node();
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

void inorder( node *root){
  if (root != NULL){
    inorder(root->left);
    cout << root->key << endl;
    inorder(root->right);
  }
}
```

```
node* insert(node* node, int key){
  if (node == NULL) return newNode(key);
  if (key < node->key)
      node->left = insert(node->left, key);
  else if (key > node->key)
      node->right = insert(node->right, key);
  return node;
}

int main(){
  struct node *root = NULL;
  root = insert(root, 50);
  insert(root, 30);
  insert(root, 20);
  insert(root, 40);
  insert(root, 70);
  insert(root, 60);
  insert(root, 80);
  inorder(root);
  system("pause");
  return 0;
}
```

FIND THE MINIMUM AND MAXIMUM IN THE BST

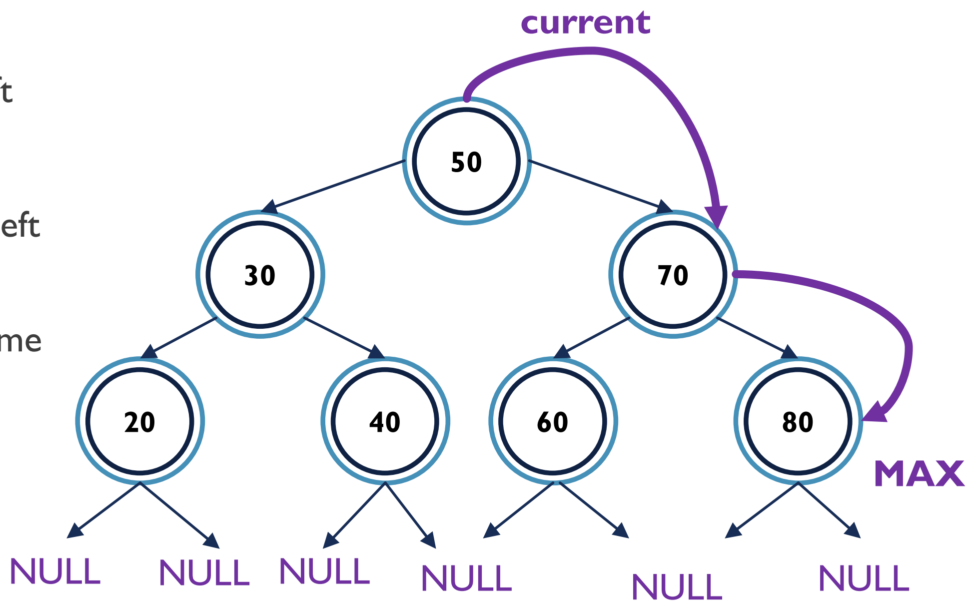
```
node* minValueNode(node* nn)
{
  node* current = nn;
  while (current->left != NULL)
     current = current->left;
  return current;
}
```



Algorithm

- From BST property the left key is always lower than current key. Create temp pointer that jumps to the left till the last leaf
- Max finding done in the same way but to the opposite direction

```
node* maxValue(node* nn) {
  node* current = nn;
  while (current->right != NULL)
    current = current->right;
  return current;
}
```



DELETE THE NODE

Case 1:

Algorithm

Node to be deleted is leaf:
Simply remove from the tree

DELETE THE NODE

Case 2:

Algorithm

Node to be deleted has only one child:
Copy the child to the node and delete the child

DELETE THE NODE

Case 3:

Algorithm

Node to be deleted has two childresn:
Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used

inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

DELETE THE KEY

```
node* deleteNode(struct node* root, int key)
{
   if (root == NULL) return root;
   if (key < root->key)
      root->left = deleteNode(root->left, key);
    else if (key > root->key)
      root->right = deleteNode(root->right, key);
    else{
      if (root->left == NULL){
          node *temp = root->right;
          delete root;
          return temp;
      }
      else if (root->right == NULL){
          node *temp = root->left;
          delete root;
          return temp;
      }

  /* node with two children: Get the inorder
successor   (smallest in the right subtree)*/

  node* temp = minValueNode(root->right);

  // Copy the inorder successor's content to this node
  root->key = temp->key;

  // Delete the inorder successor
  root->right = deleteNode(root->right, temp->key);
  }
return root;
}
```

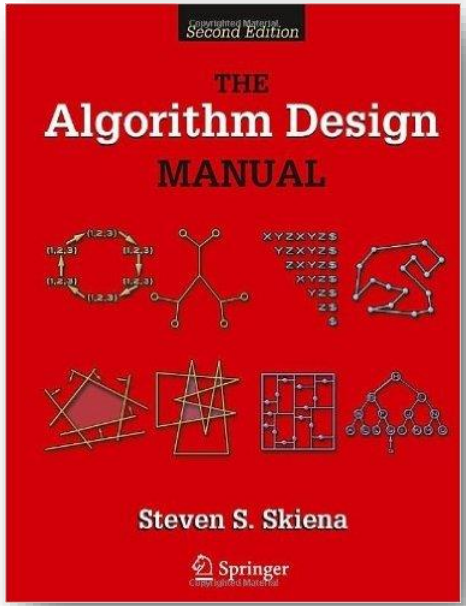
DELETE THE KEY

```cpp
int main(){
  node *root = NULL;
  root = insert(root, 50);
  root = insert(root, 30);
  root = insert(root, 20);
  root = insert(root, 40);
  root = insert(root, 70);
  root = insert(root, 60);
  root = insert(root, 80);
  cout<<"Inorder traversal of the given tree \n";
   inorder(root);
  cout<<"\nDelete 20\n";
  root = deleteNode(root, 20);
  cout<<"Inorder traversal of the modified tree \n";
   inorder(root);
  cout<<"\nDelete 30\n";
  root = deleteNode(root, 30);
  cout<<"Inorder traversal of the modified tree \n";
   inorder(root);
  cout<<"\nDelete 50\n";
  root = deleteNode(root, 50);
  cout<<"Inorder traversal of the modified tree \n";
   inorder(root);
  system("pause");
  return 0;
}
```

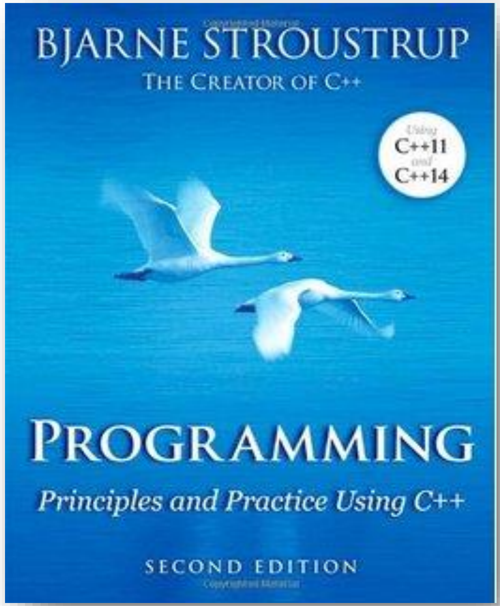
c:\users\sul\documents\visual studio 2015\P...

```
Inorder traversal of the given tree
20
30
40
50
60
70
80

Delete 20
Inorder traversal of the modified tree
30
40
50
60
70
80

Delete 30
Inorder traversal of the modified tree
40
50
60
70
80

Delete 50
Inorder traversal of the modified tree
40
60
70
80
Press any key to continue . . .
```

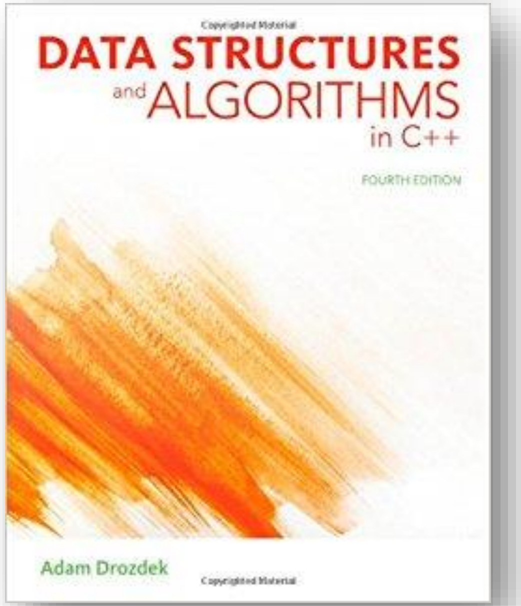
LITERATURE



Stieven Skienna
Algorithms design manual



Bjarne Stroustrup
Principles and practice using C++
Chapter 17: vectors and free store
Page 569.



Adam Drozdek
Data structures and Algorithms in C++