UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 9

Yichao (Joey)

May 30, 2020

Credit: Carey Nachenberg, Junheng Hao

Outline Today

UCLA Samueli Computer Science

- Hash Table
- Tree
- Binary Search Tree

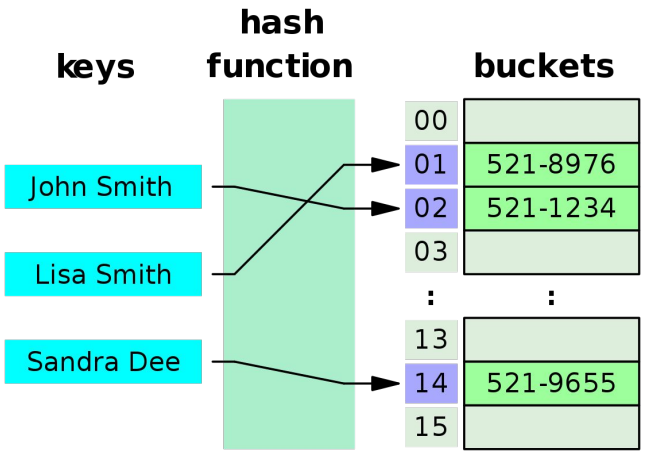
Announcements

UCLA Samueli Computer Science

- Project 4 is due 11:00 PM Thursday, June 4.

Hash Tables

UCLA Samueli Computer Science



In computing, a **hash table** (**hash map**) is a data structure that implements an **associative array abstract data type**, a structure that can **map keys to values.** A hash table uses a hash function to compute an *index*, also called a *hash code*, into an array of *buckets* or *slots*, from which the desired value can be found.

Hash Tables

UCLA Samueli Computer Science

The idea of hashing is to distribute the entries (key/value pairs) across an array of *buckets*. Given a key, the algorithm computes an *index* that suggests where the entry can be found:

```
index = f(key, array_size)
```

Often this is done in two steps:

```
hash = hashfunc(key)
index = hash % array_size
```

In this method, the *hash* is independent of the array size, and it is then *reduced* to an index (a number between `0` and `array_size - 1`) using the modulo operator (`%`).

Hash Tables

Open vs Closed Addressing

UCLA Samueli Computer Science

Open Addressing

Also known as **closed hashing**.

Collisions are dealt with by searching for **another empty buckets** within the hash table array itself.

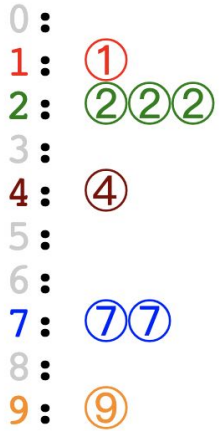
Benefits:

- Better memory locality and cache performance. All elements laid out linearly in memory.
- Performs better than closed addressing when the number of keys is known in advance and the churn is low.

Closed Addressing

Also known as **open hashing**.

A key is always stored in the bucket it's hashed to. Collisions are dealt with using **separate data structures** on a per-bucket basis.



Benefits:

- Easier removal (no need for deleted markings)
- Typically performs better with high load factor.

Hash Tables

Operations

UCLA Samueli Computer Science

- Insert
- Remove
- Search

- The complexity depends on your hash tables.
- Usually,

Algorithm	**Average**	**Worst case**
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Hash Tables

UCLA Samueli Computer Science

Pretty much any time you want to map keys to values with constant time for lookup/add/remove. If you find yourself **looping through a list to find an element**, think if there's a way to store the elements with keys in **a hash table** (aka hash map, aka dictionary) instead.

```
1.  class Contacts {
2.   List<Person> contacts = new ArrayList<Person>();
3.
4.   void addContact(Person p) {
5.     contacts.add(p);
6.   }
7.   // O(n) time
8.   Person getContact(String phoneNumber) {
9.     for (Person p : contacts) {
10.      if (p.getPhoneNumber().equals(phoneNumber))
     return p;
11.    }
12.    return null;
13.  }
14.  ...
15. }
```

```
1.  class Contacts {
2.   List<Person> contacts = new ArrayList<Person>();
3.   Map<String, Person> phoneNumberToPerson = new
     HashMap<String, Person>();
4.
5.   void addContact(Person p) {
6.     contacts.add(p); // assuming this list is still
     needed somewhere else
7.     phoneNumberToPerson.put(p.getPhoneNumber(), p);
8.   }
9.
10.  // O(1) time!
11.  Person getContact(String phoneNumber) {
12.    return phoneNumberToPerson.get(phoneNumber);
13.  }
14.  ...
15. }
```

Hash Tables

Simple question

UCLA Samueli Computer Science

Que – 1. Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true?

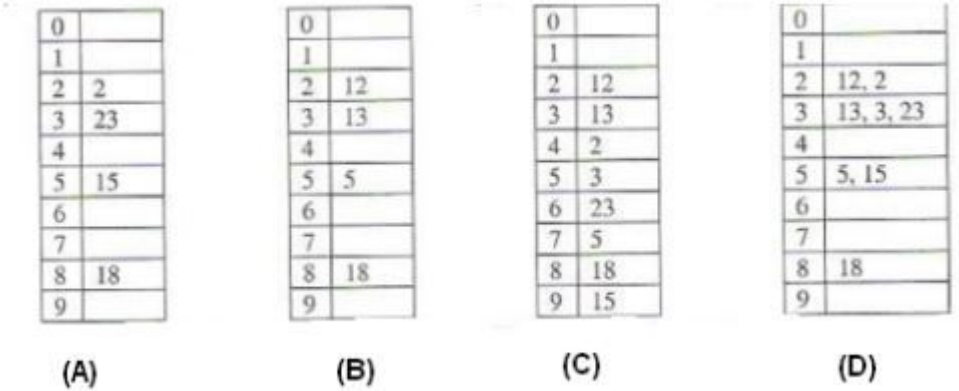
i. 9679, 1989, 4199 hash to the same value
ii. 1471, 6171 has to the same value
iii. All elements hash to the same value
iv. Each element hashes to a different value
(A) i only
(B) ii only
(C) i and ii only
(D) iii or iv

Hash Tables

Simple question

UCLA Samueli Computer Science

Que – 2. The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?



0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

Hash Tables

Simple question

UCLA Samueli Computer Science

Que – 3. A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

(A) 46, 42, 34, 52, 23, 33
(B) 34, 42, 23, 52, 33, 46
(C) 46, 34, 42, 23, 52, 33
(D) 42, 46, 33, 23, 34, 52

Hash Tables

FNV-1

UCLA Samueli Computer Science

- Hash functions: Take a “key” and map it to a number
- Requirement for hash function: should return the same value for the same key
- Good hash functions:
  - Spreads out the values: two different key are likely to results in different hash values. → **Avoid confliction**
  - Compute each value quickly.
- Example: **FNV-1**

Fowler–Noll–Vo (FNV) is a non-cryptographic hash function created by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo.
http://www.isthe.com/chongo/tech/comp/fnv/#FNV-param

```
unsigned int FNV-1(string s) {
  unsigned int h = 2166136261U;
  for (int k = 0; k != s.size(); k++)
  {
    h += s[k];
    h *= 16777619;
  }
  return h;
}
```

offset_basis

FNV_prime

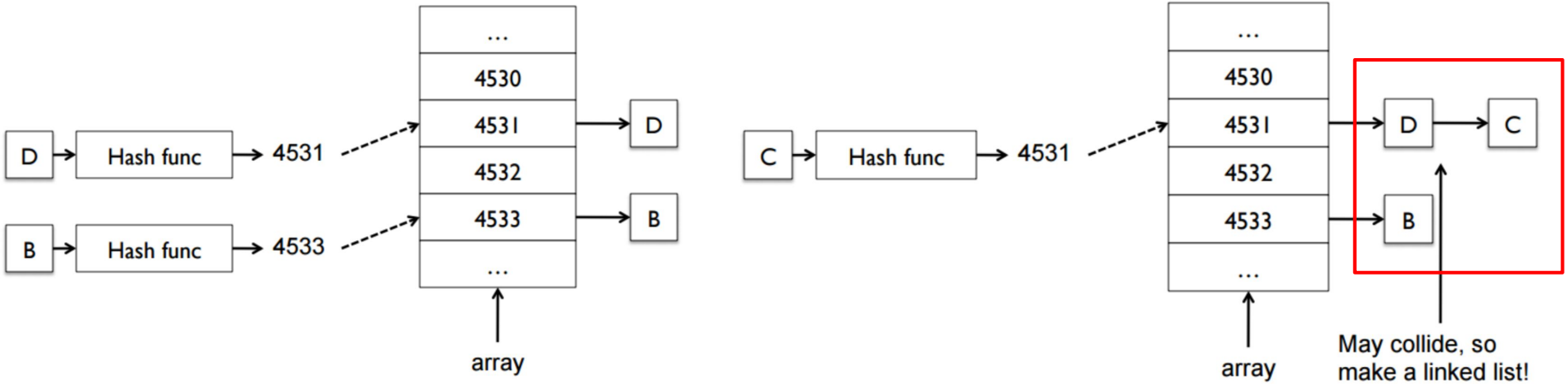
Hash Tables

Examples

UCLA Samueli Computer Science

- Example: Use a hash table to store people.
- Use a linked list to collision in the hash function.

*"You should almost **NEVER** assume that collisions are impossible!!!" --David Smallberg*



Hash Tables Problem

The very first question in LeetCode - TwoSum

UCLA Samueli Computer Science

- Given an array of integers, return indices of the two numbers such that they add up to a specific target.
- You may assume that each input would have exactly one solution, and you may not use the same element twice.
- Example: Given `nums = [2, 11, 7, 15]` and `target = 9`. Because `nums[0] + nums[2] = 2 + 7 = 9`, return `[0, 2]`.

1. Two Sum

Hash Table in STL

Map, multimap, unordered_map, HashMap

UCLA Samueli Computer Science

- Useful functions of STL `map`, `multimap`, `unordered_map`
  - `size(), begin(), end(), empty()`
  - `insert(keyvalue, mapvalue)`
  - `find()`
  - `operator[]`
- Internal implementation:
  - `map/multimap`: Red-black tree
  - **`unordered_map`**: Hash table

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> m;
        for(int i = 0; i < nums.size(); i++)
        {
            if(m.find(target-nums[i]) != m.end())
                return {m[target-nums[i]], i};
            m[nums[i]] = i;
        }
    }
 };
```

Tree

Definition

UCLA Samueli Computer Science

- Terms: Node/edge, root node, leaf node, parent and child node, subtree, levels (height/depth).
- Features: No loop, no shared children
- Question: How many edges should there be in a tree with ***n*** nodes?
- Binary tree: no node has more than two children.
- Question: How many nodes can a binary tree of height ***h*** have? → Full binary tree

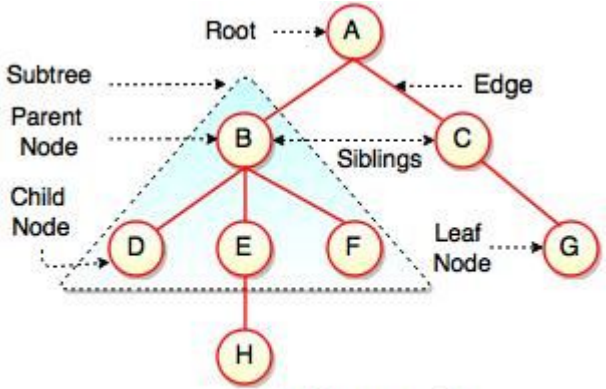
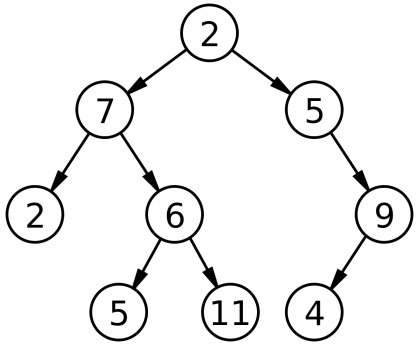


Fig. Structure of Tree



Tree

As a data structure

UCLA Samueli Computer Science

- Tree is a useful data structure!
- Basic functions: insert, remove, search, **traverse**
- How to traverse a tree?

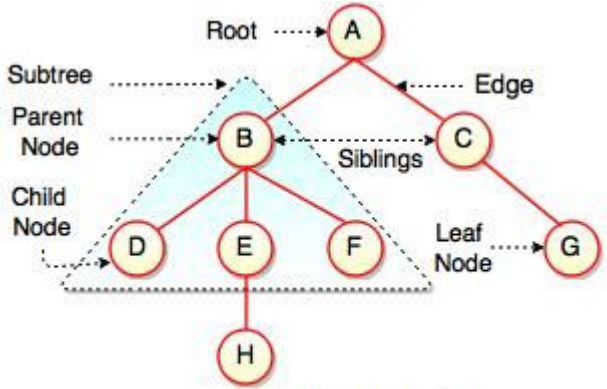
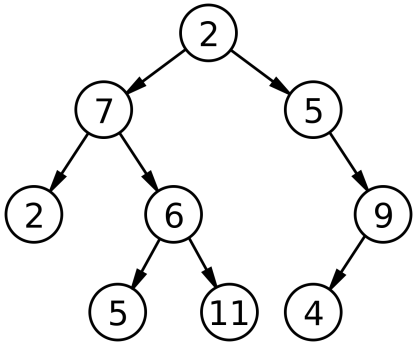


Fig. Structure of Tree

```
struct Node{
  ItemType val;
  Node* leftChild;
  Node* rightChild;
} // a simple node
```

```
Class Tree{
public:
  // ???
private:
  // ???
}
```



Tree Traversal: Pre-order

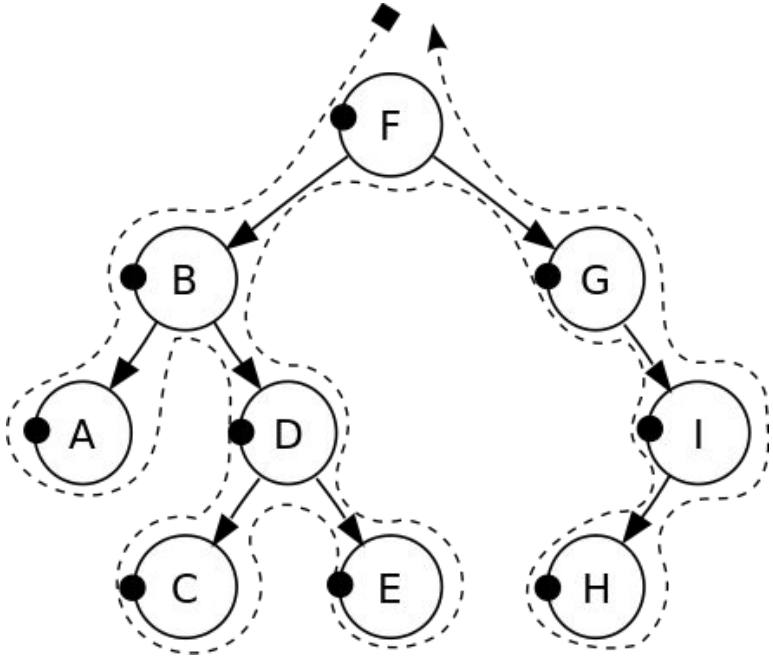
Three methods of tree traversal

UCLA Samueli Computer Science

```
void preorder(const Node* node)
{
  if (node == nullptr) return;
  cout << node->val << “,”;
  preorder(node->left);
  preorder(node->right);
}
```

Pre-order output:

`F, B, A, D, C, E, G, I, H`



Tree Traversal: In-order

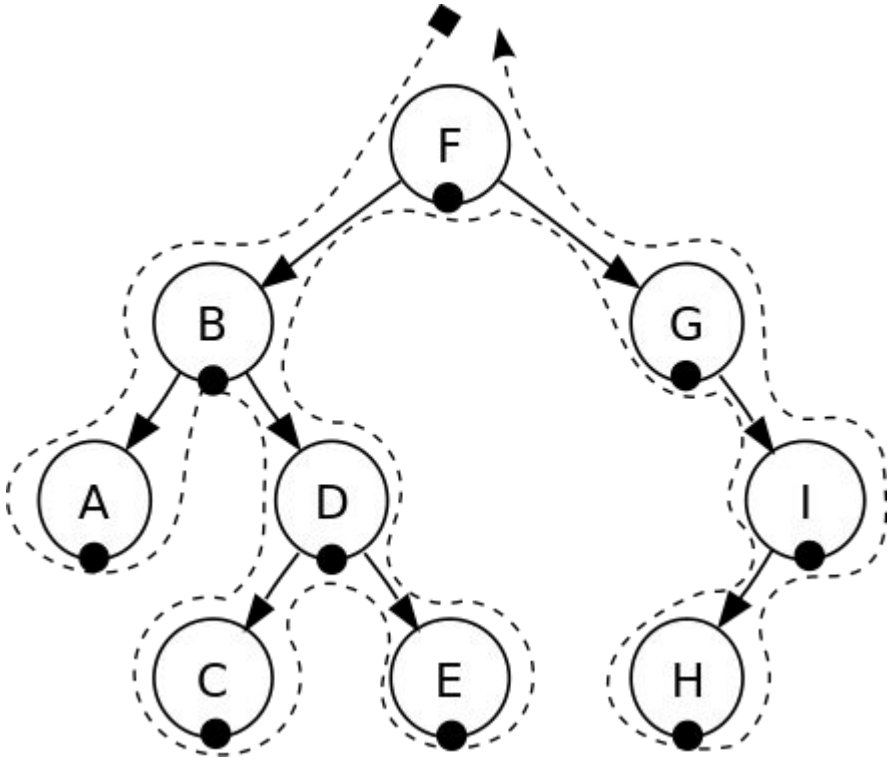
Three methods of tree traversal

UCLA Samueli Computer Science

```
void inorder(const Node* node)
{
  if (node == nullptr) return;
  inorder(node->left);
  cout << node->val << “,”;
  inorder(node->right);
}
```

In-order output:

`A, B, C, D, E, F, G, H, I`



Tree Traversal: Post-order

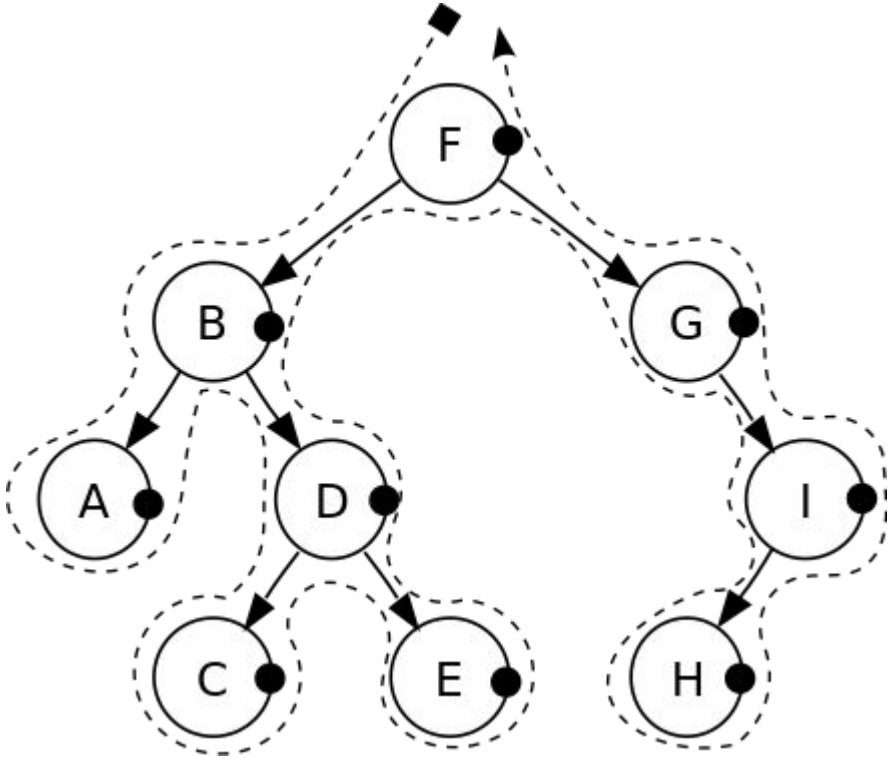
Three methods of tree traversal

UCLA Samueli Computer Science

```
void postorder(const Node* node)
{
  if (node == nullptr) return;
  postorder(node->left);
  postorder(node->right);
  cout << node->val << ",";
}
```

Post-order output:

`A, C, E, D, B, H, I, G, F`



Tree Traversal: Compare

UCLA Samueli Computer Science

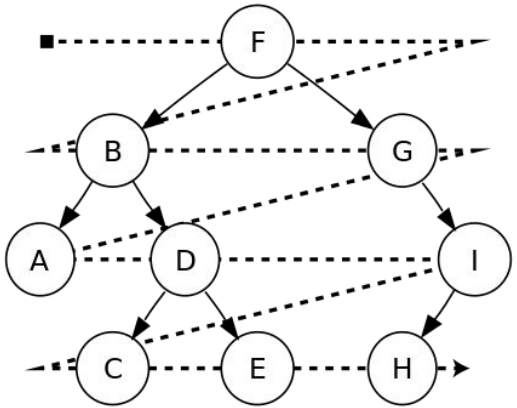
```
void preorder(const Node* node)
{
  if (node == nullptr) return;
  cout << node->val << ",";
  preorder(node->left);
  preorder(node->right);
}
```

```
void inorder(const Node* node)
{
  if (node == nullptr) return;
  inorder(node->left);
  cout << node->val << ",";
  inorder(node->right);
}
```

```
void postorder(const Node* node)
{
  if (node == nullptr) return;
  postorder(node->left);
  postorder(node->right);
  cout << node->val << ",";
}
```

//Other ways?

Level-order or say breadth-first search!



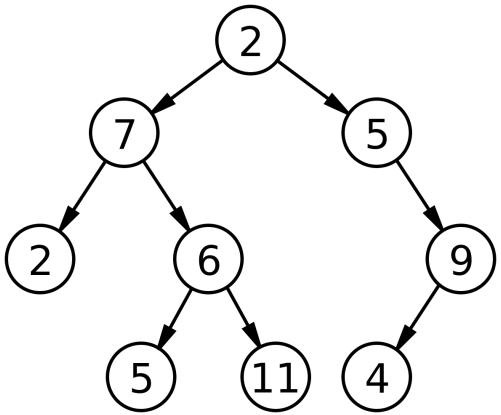
Tree

Recursion in trees

UCLA Samueli Computer Science

- It is easy and natural to apply recursion on trees!
- Pre-order / in-order / post-order are all recursive methods to traverse a tree.
- Question: How to calculate a height of a given tree?

```
int getTreeHeight(const Node* node)
{
  if (node == nullptr) return 0;
  int leftHeight = getTreeHeight(node->left);
  int rightHeight = getTreeHeight(node->right);
  if (leftHeight > rightHeight)
    return leftHeight + 1;
  else
    return rightHeight + 1;
}
```

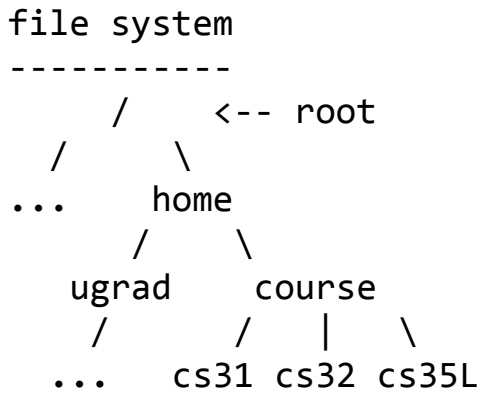


Tree

Conclusions and applications

UCLA Samueli Computer Science

- Non-linear data structures unlike arrays or lists.
- Present hierarchy
- Optimized (like balanced) tree and variants can improve efficiency of search, sort and other typical problems.
- Other tree variants:
  - Binary search Tree
  - B tree / B+ tree
  - Red-black tree
  - K-D Tree
  - Suffix Tree

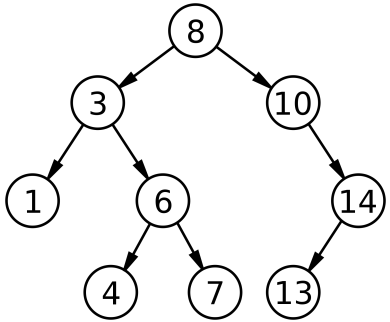


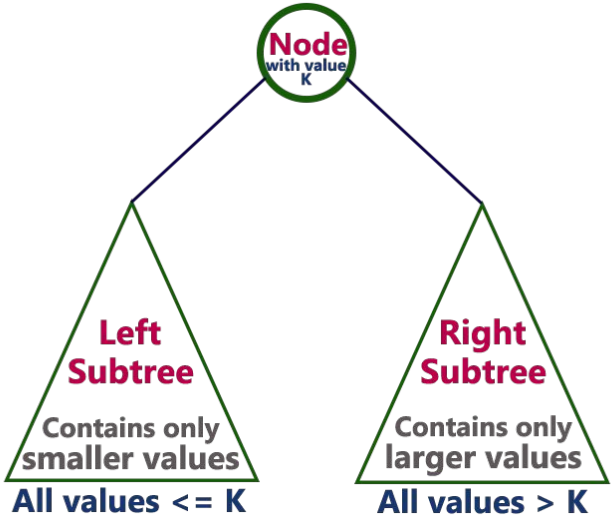
Binary Search Tree

Definition, Properties

UCLA Samueli Computer Science

- The left subtree of a node contains only nodes with keys lesser 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.





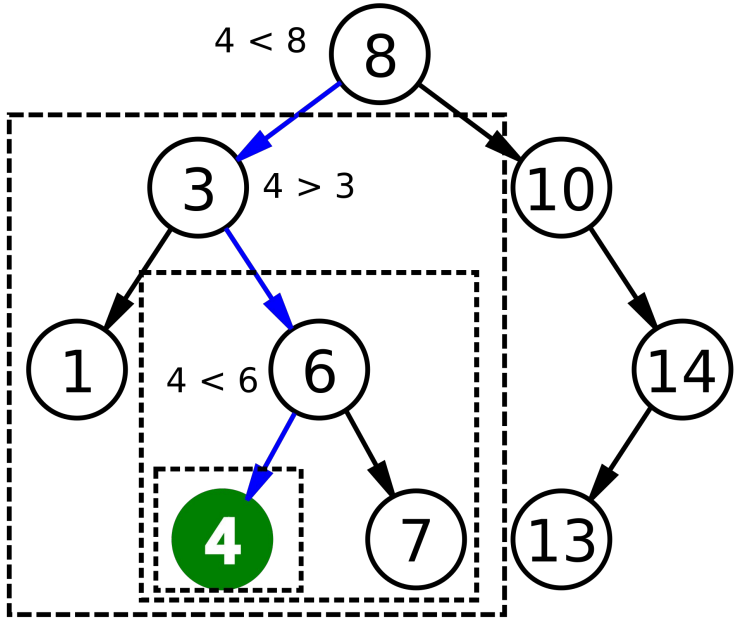
Binary Search Tree

Search

UCLA Samueli Computer Science

```
node* search(node* node, ItemType key)
{
    /* If the tree is empty, return null pointer */
    if (node == nullptr) return nullptr;

    /* compare with current node and decide*/
    if (key == node->key)
        return node;
    else if (key < node->key)
        return search(node->left, key);
    else
        return (node->right, key);
}
```



Binary Search Tree

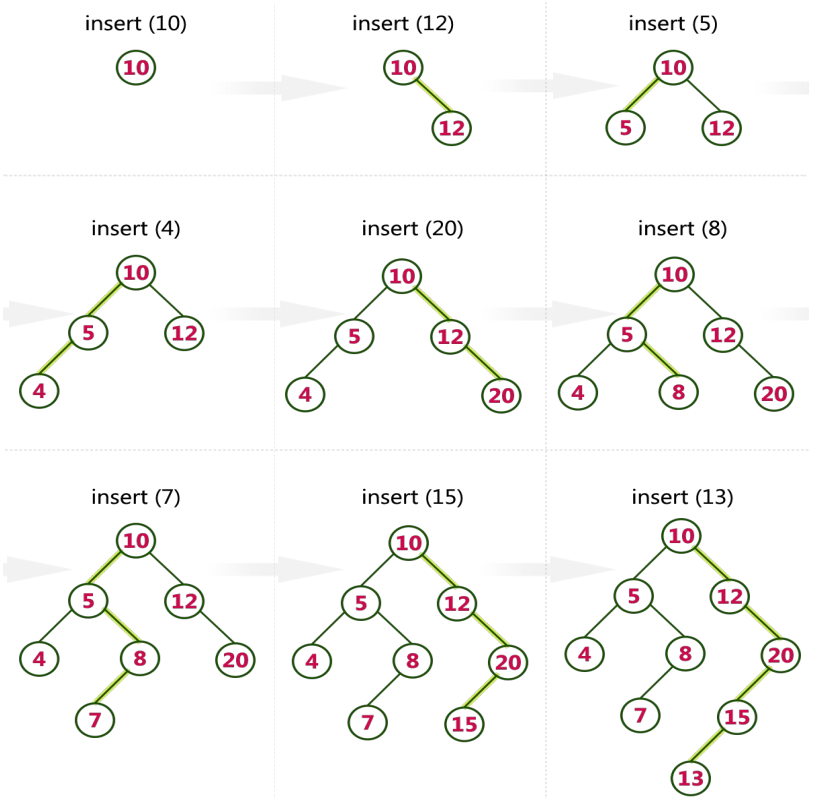
Insertion

UCLA Samueli Computer Science

```
node* insert(node* node, ItemType key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```



Binary Search Tree

Deletion

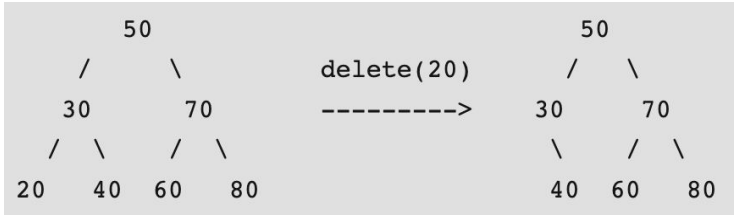
UCLA Samueli Computer Science

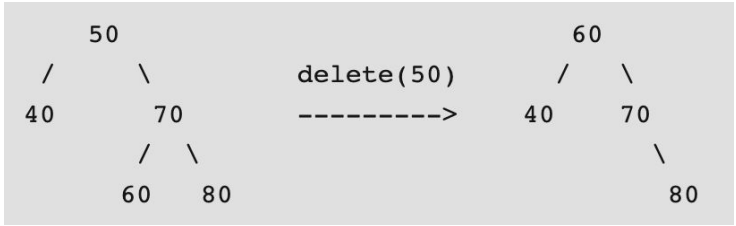
- Deletion is the most tricky one.
- 3 cases:
  - The node is a leaf node.
  - The node has one child.
  - The node have two children.

Super easy! Just delete that node!

Not difficult! Copy the child to the node and delete the child.

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.





Binary Search Tree

Deletion

UCLA Samueli Computer Science

```
node* delete(node* node, ItemType key)
{
  if (node == nullptr) return nullptr;
  if (key < node->key) { node->left = delete(node->left, key); }
  else if (key > node->key) { node->right = delete(node->right, key); }
  else{
    /* case 1 & case 2*/
    if (node->left == nullptr) { node *temp = node->right; delete(node); return temp; }
    else if (node->right == nullptr) {node *temp = node->left; delete(node); return temp;}
    /* case 3 */
    node* temp = minValueNode(node->right);
    node->key = temp->key;
    root->right = deleteNode(root->right, temp->key);
  }
}
```

Binary Search Tree

Analysis of BST

UCLA Samueli Computer Science

- Insertion
  - The (worst) case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node.
  - The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).
  - Average time complexity: O(log *n*)
- Deletion
  - Similar to insertion for complexity analysis

Binary Search Tree

FindMin and FindMax by using recursion

UCLA Samueli Computer Science

Question: How to test a tree is a valid BST?

One possible recursion solution by using **`findMinKey`** and **`findMaxKey`**.

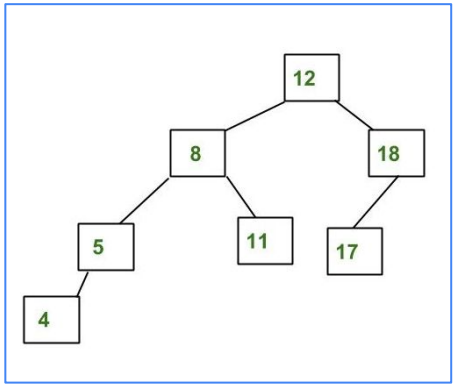
```
bool isValidBST(const node* node)
{
  if (node == nullptr) return true;
  /* check left subtree and right subtree condition*/
  if (node->left != nullptr && findMaxKey(node->left) > node->key)
    return false;
  if (node->right != nullptr && findMinKey(node->right) < node->key)
    return false;
  /* further check subtree with left child and right child */
  return isValidBST(node->left) && isValidBST(node->right)
}
```

Beyond Binary Search Tree

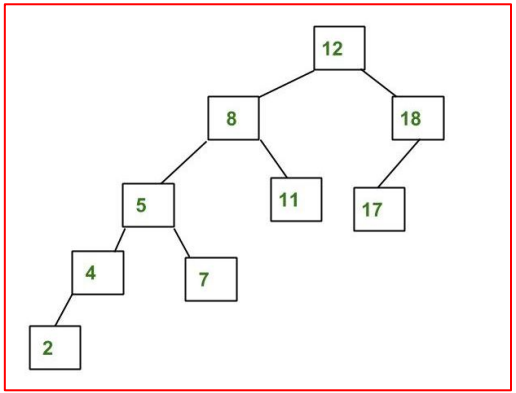
AVL Tree

UCLA Samueli Computer Science

- What is the drawback of naive BST? → It can be skewed! Not good!
- AVL (Adelson-Velsky and Landis) Tree is a self-balancing BST.



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

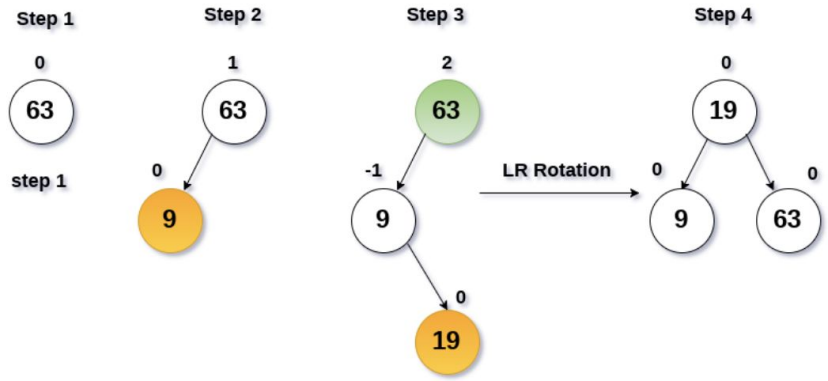
Beyond Binary Search Tree

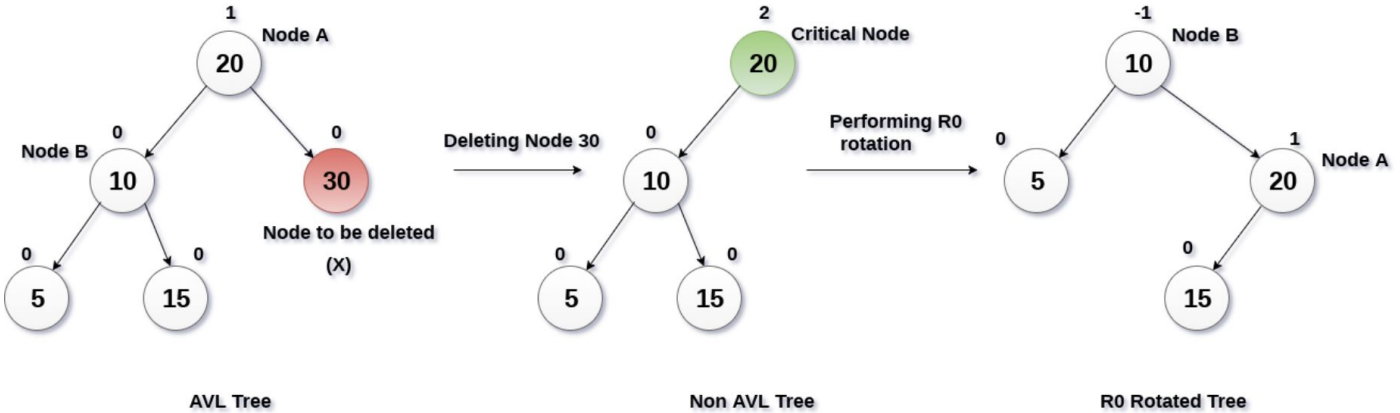
AVL Tree: Operations

UCLA Samueli Computer Science

- Insertion
- Deletion

Please check this interesting demo!





Beyond Binary Search Tree

The tree family (1)

UCLA Samueli Computer Science

There are many interesting tree structures such as:

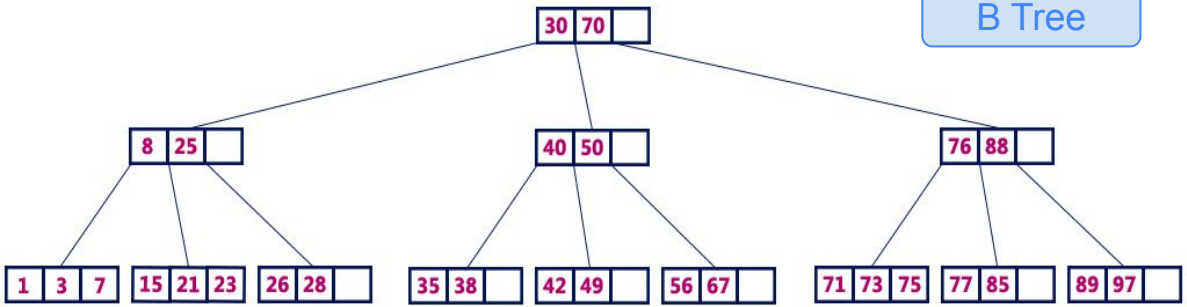
- B tree and B+ tree
- 2-3-4 tree
- R tree (spatial index tree)
- Red-black tree
- K-D tree

Beyond Binary Search Tree

The tree family (2)

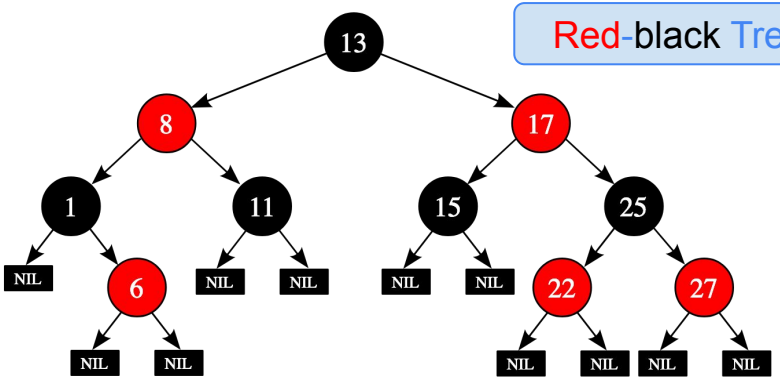
UCLA Samueli Computer Science

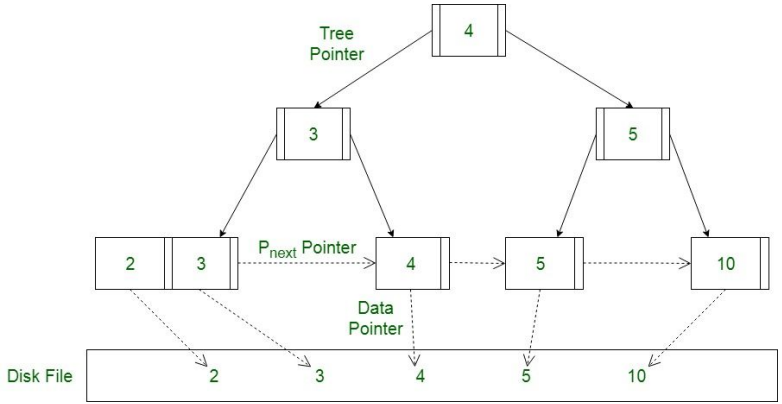
B Tree



B+ Tree

Red-black Tree



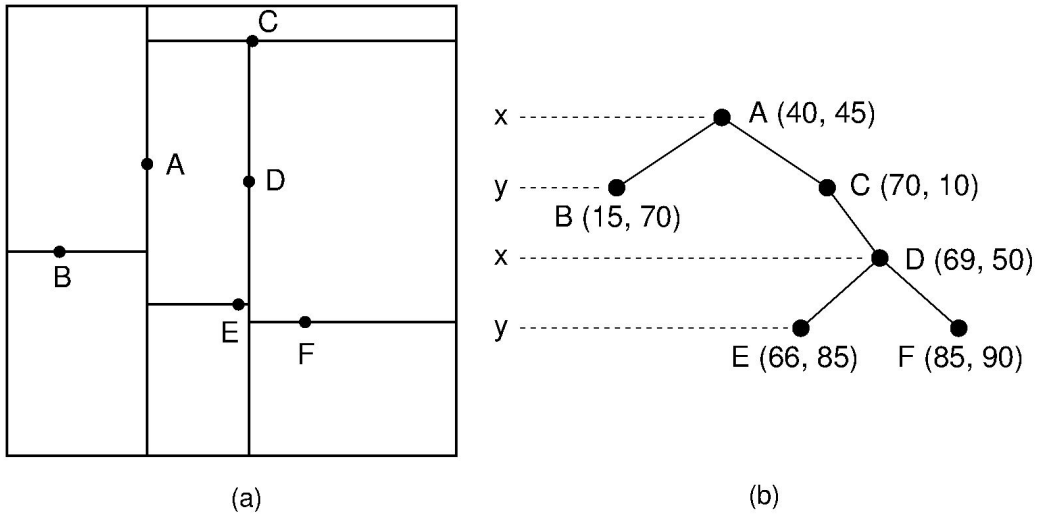


Beyond Binary Search Tree

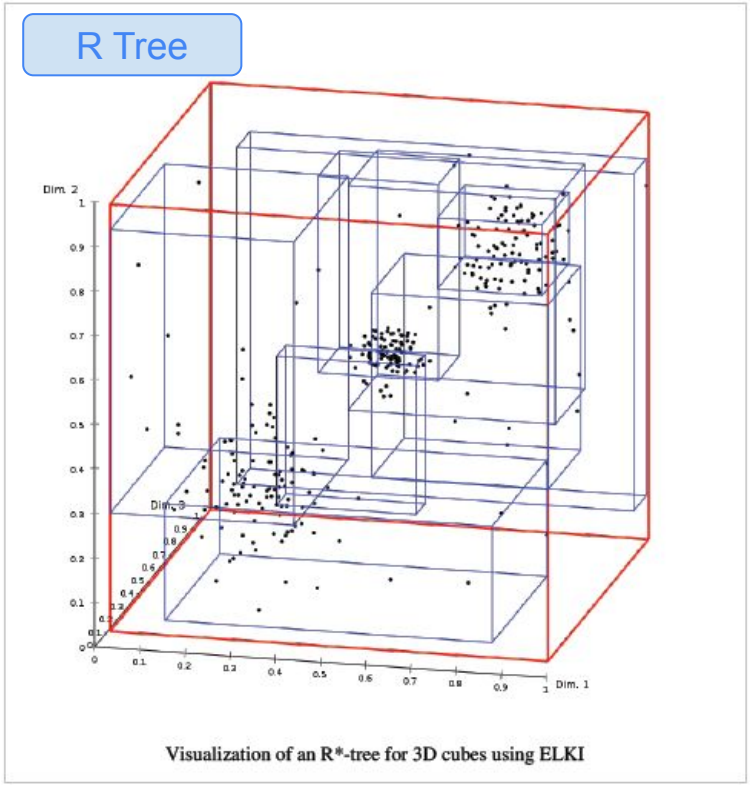
The tree family (3)

UCLA Samueli Computer Science

KD Tree



R Tree



Visualization of an R*-tree for 3D cubes using ELKI