# CS32: Introduction to Computer Science II
# **Discussion Week 9**

Junheng Hao, Arabelle Siahaan
May 31, 2019

# Announcements

- Homework 5 is due 11PM next Tuesday, June 5. (@A@)

- Project 4 is also due 11PM next Tuesday, June 5. (@A@ >_< @A@)

# Outline Today

- Binary Search Tree

- Heap (Preview)

- Hash Tables

- Project 4 Guidelines

# Binary Search Tree
Definition, Properties

- 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.
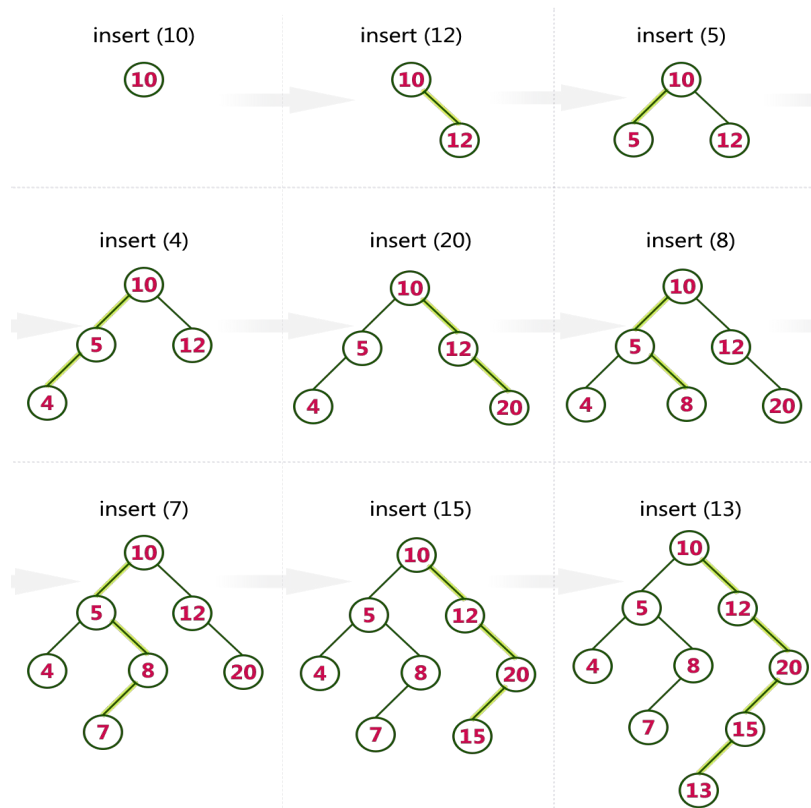- The left and right subtree each must also be a binary search tree.
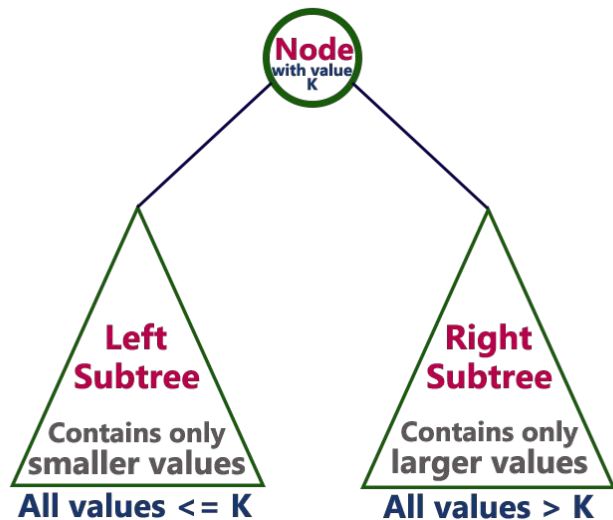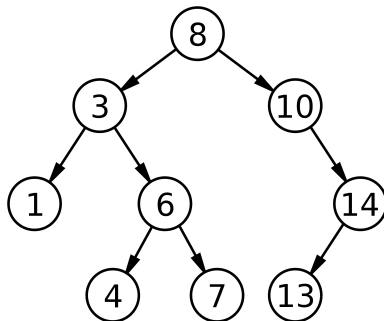
# Binary Search Tree

Insertion

```
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;
}
```

insert (10)   insert (12)   insert (5)

insert (4)   insert (20)   insert (8)

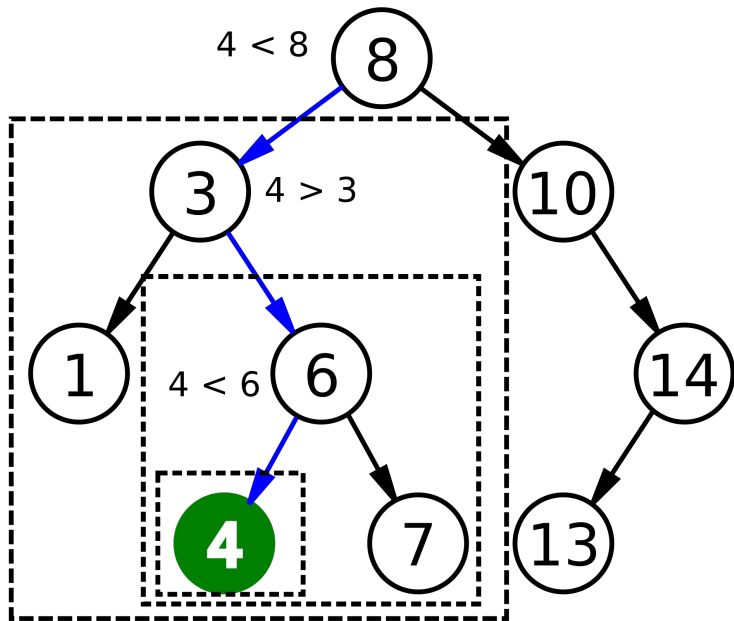insert (7)   insert (15)   insert (13)

# Binary Search Tree
Search

```
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);
}
```



4 < 8

8

3    4 > 3

10

1    4 < 6    6

4    7    13

14

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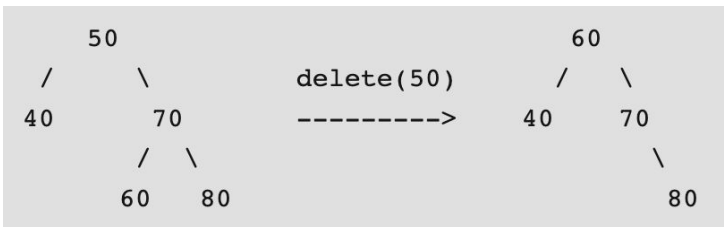- 3 cases:
  - The node is a leaf node.
  - The node has one child.
  - The node have two children.

*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.*

```
      50                                      50
    /    \         delete(20)              /    \
  30      70      --------->             30      70
 /  \    /  \                              \    /  \
20  40  60  80                             40  60  80
```

```
    50                                      50
  /    \          delete(30)              /    \
30      70       --------->             40      70
  \    /  \                                    /  \
  40  60  80                                  60  80
```

```
    50                                      60
  /    \          delete(50)              /    \
40      70       --------->             40      70
       /  \                                      \
      60  80                                      80
```

# Binary Search Tree
Deletion

```
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);
  }
}
```

- 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

How to implement **findMaxKey** and **findMinKey** function?

We assume the root is not `nullptr`.

```cpp
node* findMaxKey(const node* node)
{
  /* only need to check the right subtree*/
  if (node->right == nullptr) return node->key;
  return findMaxKey(node->right);
}
```

```cpp
node* findMinKey(const node* node)
{
  /* only need to check the right subtree*/
  if (node->left == nullptr) return node->key;
  return findMinKey(node->left);
}
```

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

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

```cpp
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)
}
```

*What is the complexity of this isValidBST() function?*

# Beyond Binary Search Tree
AVL Tree

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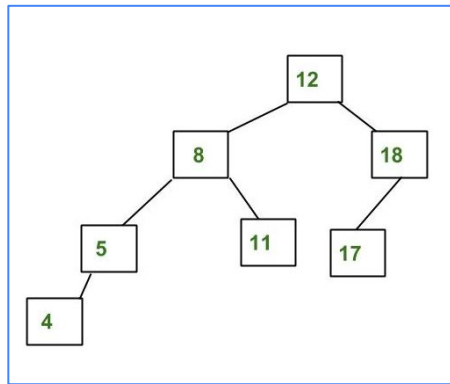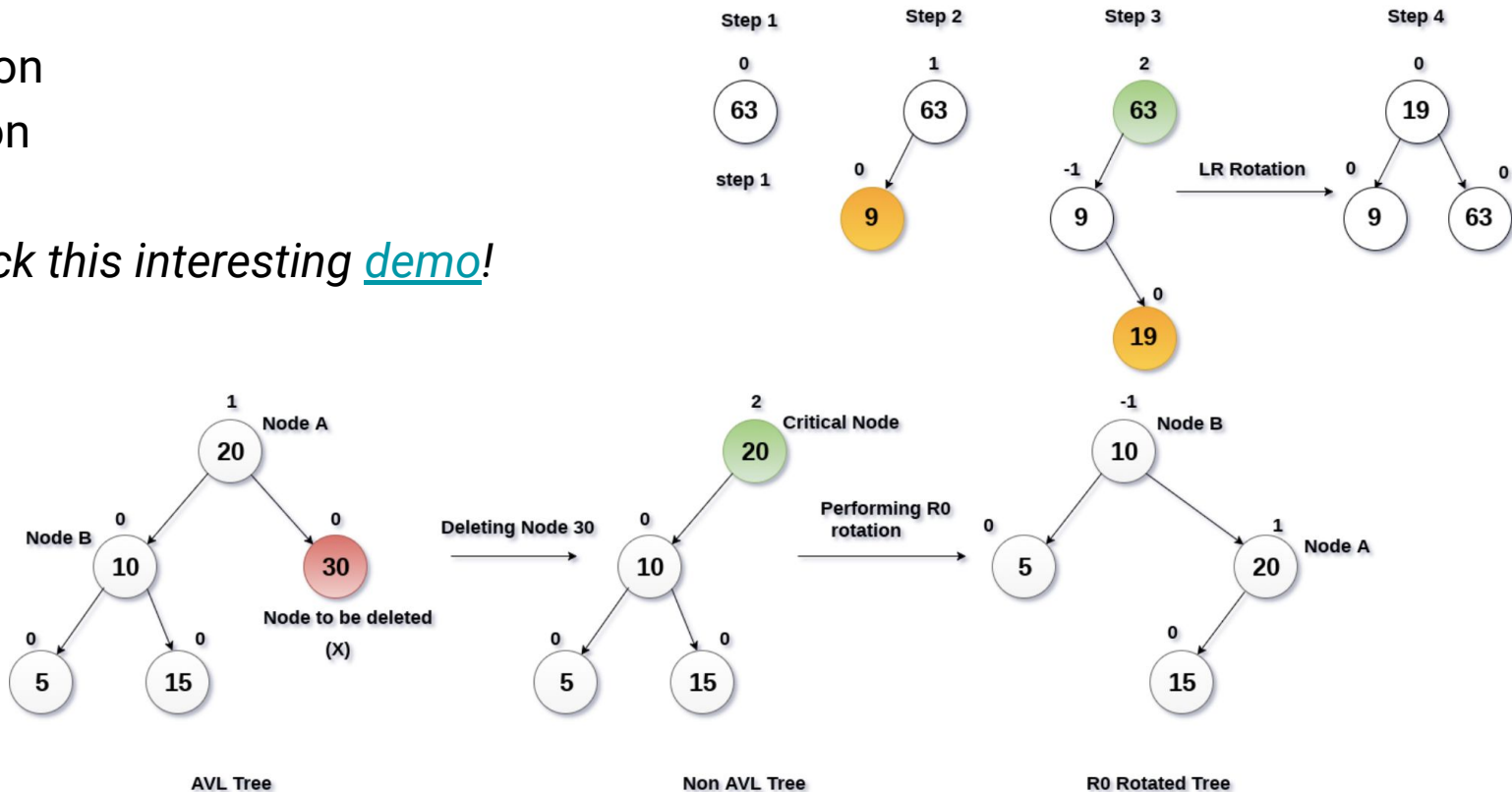


*This is OK!*
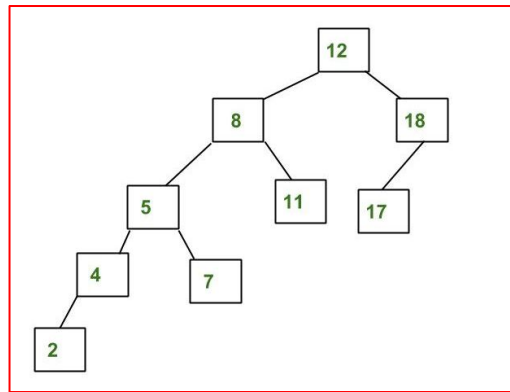


*Just OK is not OK!*

# Beyond Binary Search Tree

AVL Tree: Operations

- Insertion
- Deletion

*Please check this interesting [demo](#)!*

# Beyond Binary Search Tree

The tree family (1)

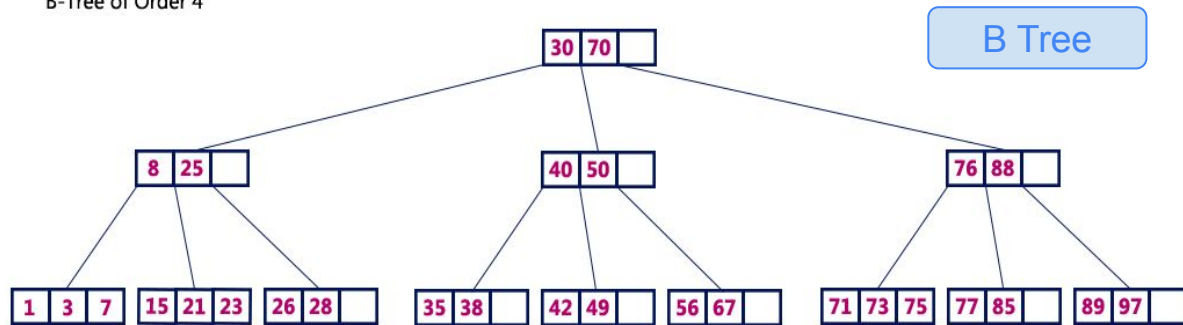There are many interesting tree structures such as:
- B tree and B+ tree (I did not hear about A+ tree though)
- 2-3-4 tree
- R tree (spatial index tree)
- **Red-black tree**
- K-D tree
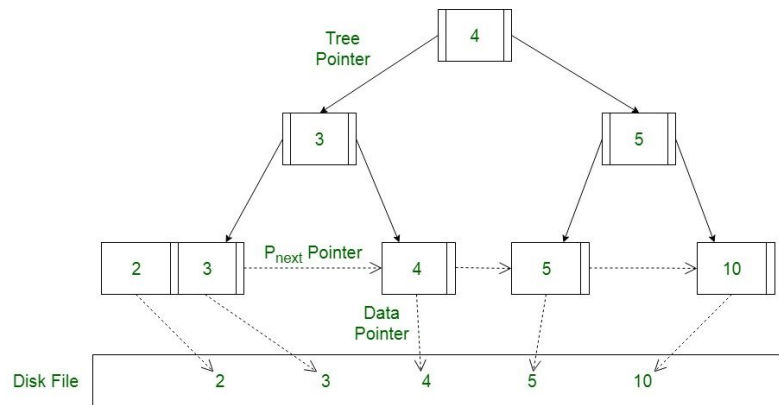
*Please check the given links for more details!*

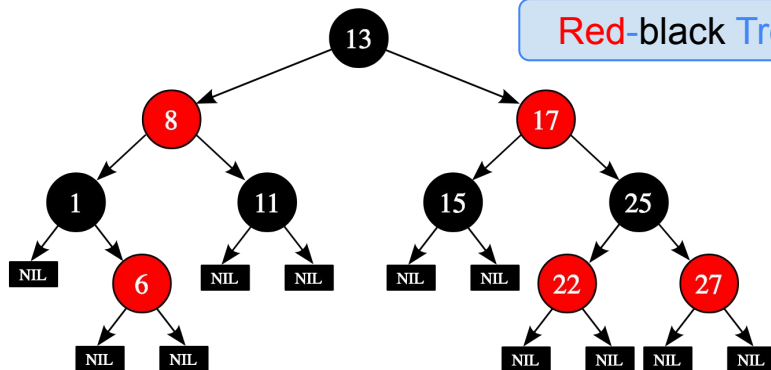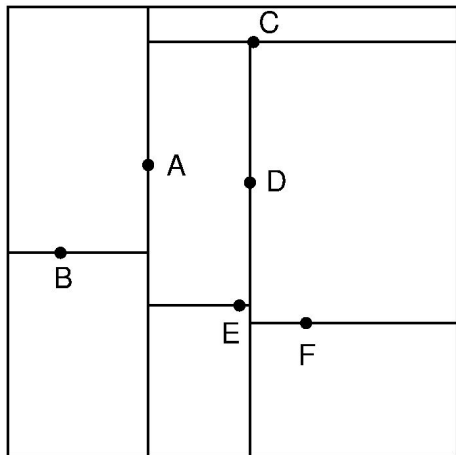# Beyond Binary Search Tree

The tree family (2)

# Beyond Binary Search Tree
The tree family (3)

KD Tree



(a)

A (40, 45)
x

B (15, 70)
y

C (70, 10)

D (69, 50)
x
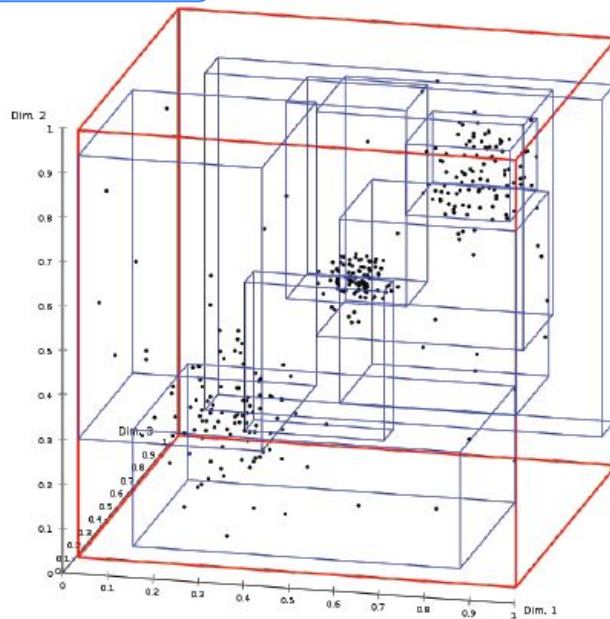
E (66, 85)     F (85, 90)
y
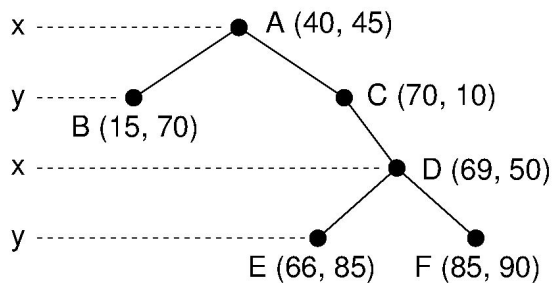
(b)

R Tree



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

# Heap (preview)
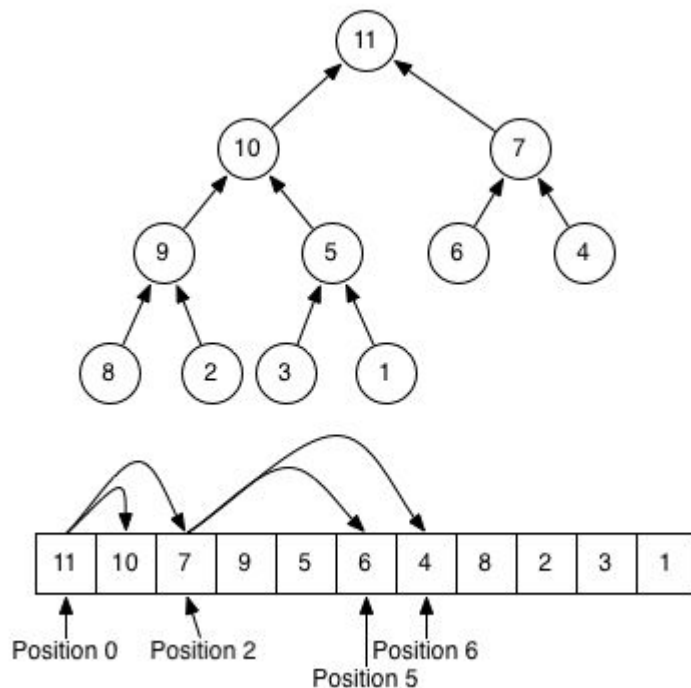## Definition and properties

- About heap
  - Heap is considered as complete binary tree.
  - Every nodes carries a value greater than or equal to its children (for MaxHeap).
  - Often implemented as an array.
  - Body structure of priority queue.



Ordered, on top of each other

No particular order

**Stack**          **Heap**
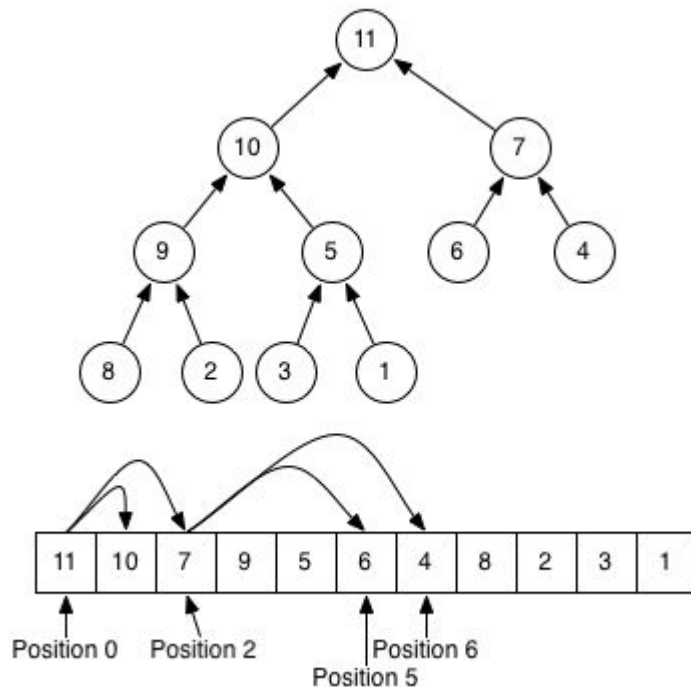


Position 0   Position 2   Position 6
Position 5

# Heap (preview)
Standard operations

- Three operations of heaps
  - Find Max (search)
  - Insert Node (insert)
  - Delete Max (delete)

- How to implement `FindMax()` function of a heap?
  - Well, that is just too obvious!

# Heap & Heapsort (preview)
## Complexity of heap operations

- Find Max → O(1)
- Insert → O(log *n*)
- Delete Max Node → O(log *n*)

6 5 3 1 8 7 2 4

- Bonus: How can you sort based on heap?
  - Insert all elements into a heap.
  - Extract the maximum element from the heap one by one.
  - Check the example in Wikipedia
- What is the complexity of heapsort?
  - $O(n \log n)$

# Hash Tables
Start from hashing and hash functions

- 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

"David Smallberg" ──→ [ Hash Function H ] ──→ 4531

```
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;
}
```

# Hash Tables
## Examples

- 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
Operations
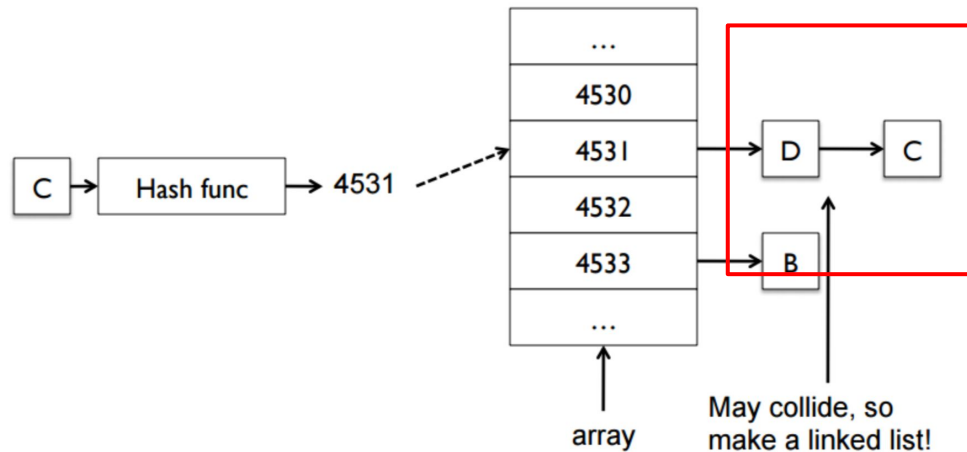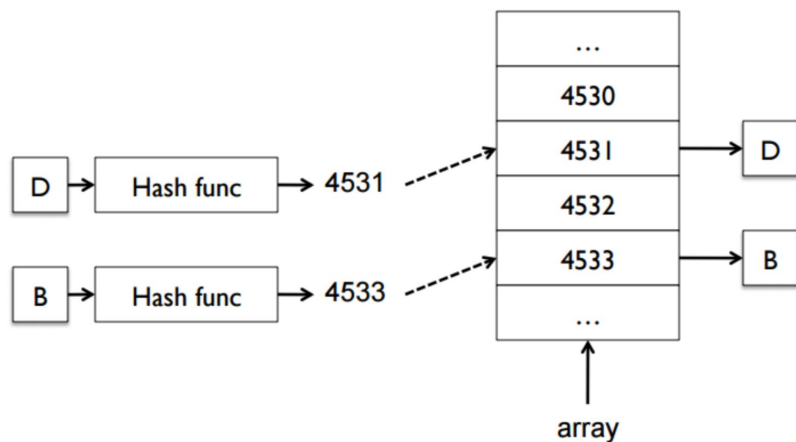
- Insert
- Remove
- Search

- The complexity depends on your hash tables.
- Closed Hashing
  - Fixed number of buckets
  - All operations are **O(n)** with a small constant of proportionality
- Open Hashing
  - Consider *#entries / #buckets*
  - Almost **O(1)** for all operations

# Hash Tables Problem 1
Word counting and sorting in a document

- Question: We have **n** words in a document, whose vocabulary size is **v**. Count top k frequent words in a document.

Two steps: counting + sorting

- The most efficient way to count the frequency for all words takes **O**(___$n$___) time complexity.

- After getting the frequency of each word, the most efficient way to get the top k frequent words takes **O**(___$v \log k$___) time complexity.

- Totally the entire procedure takes **O**(___$n + v \log k$___).

# Hash Tables Problem 2
The very first question in LeetCode - TwoSum

- 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, 7, 11, 15]` and `target = 9`. Because `nums[0] + nums[1] = 2 + 7 = 9`, return `[0, 1]`.

# Hash Table in STL
Map, multimap, unordered_map, HashMap

- 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
- Pros and cons between `map` and `unordered_map`

# Hints for Project 4

**Note & Reminders:**

1. Get File I/O done first.
2. Build BST or hashtable by `vector`, `list`, `stack` and `queue` (and `string`) instead of `map` or `unordered_map`.
   - Reason to build BST or hashtable: fast indexing the substrings
   - For complexity analysis
3. Algorithm improvements → Shorten the diff file

# Break Time! (5 minutes)

**Q & A**

# Group Exercises: Worksheet

- Exercise problems from **Worksheet 8** (see "LA worksheet" tab in CS32 website). Answers will be posted next week.

- Questions for today:

  - TBA

Content

# Thank you!

## Q & A