UCLA Samueli Computer Science

CS32: Introduction to Computer Science

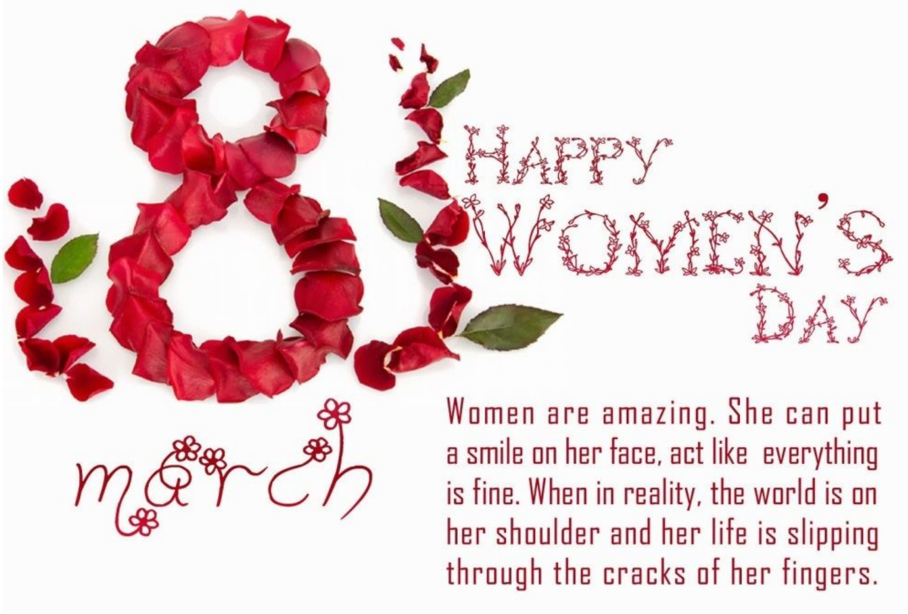
Discussion Week 9

Junheng Hao, Ramya Satish

Mar 8, 2019

Announcement

- Homework 5 and Project 4 are both due on March 14 (Tuesday).
- Final exam: March 16, 11:30-2:30



Outline

UCLA Samueli Computer Science

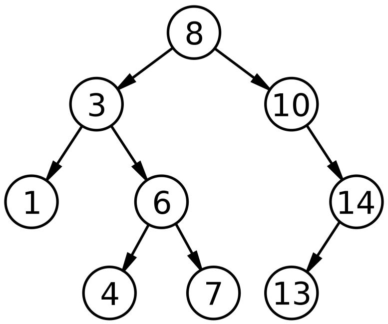
- Binary Search Tree
- Heap (Introduction)
- Hash Tables

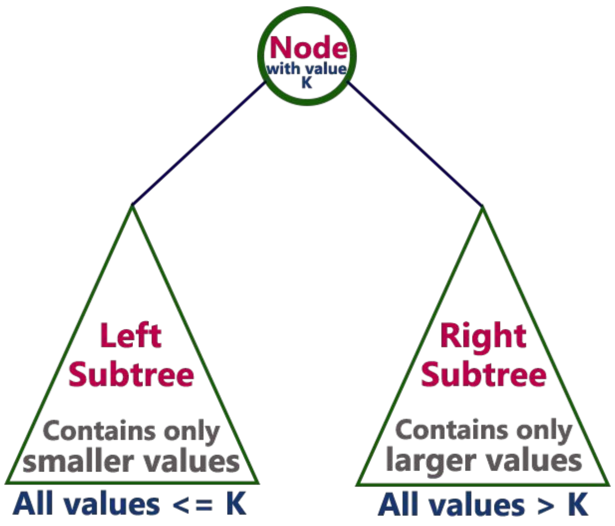
Binary Search Tree

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

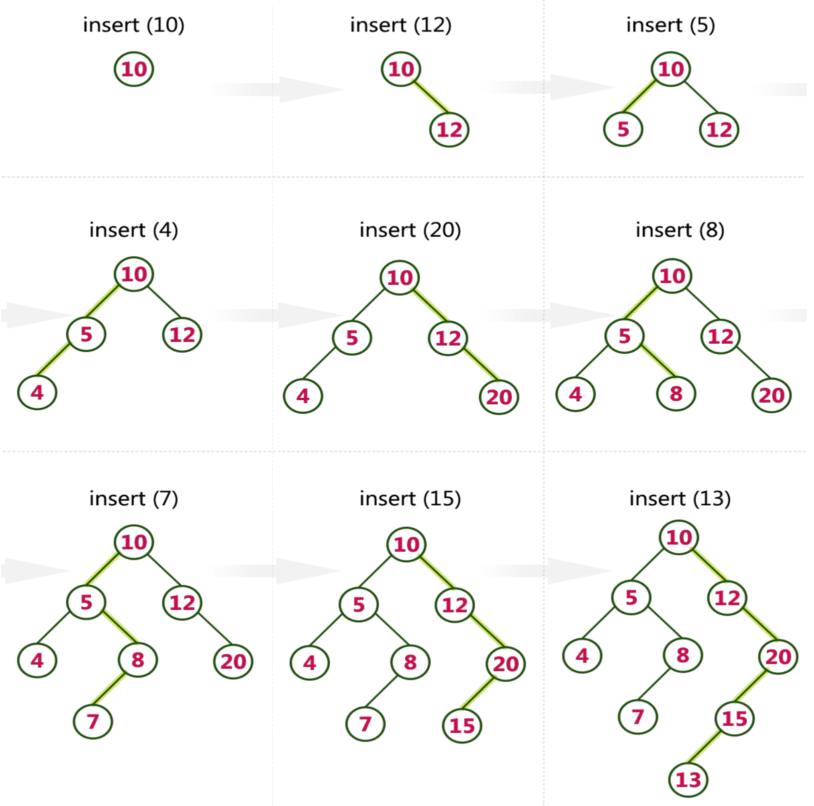
Insertion

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;
}
```

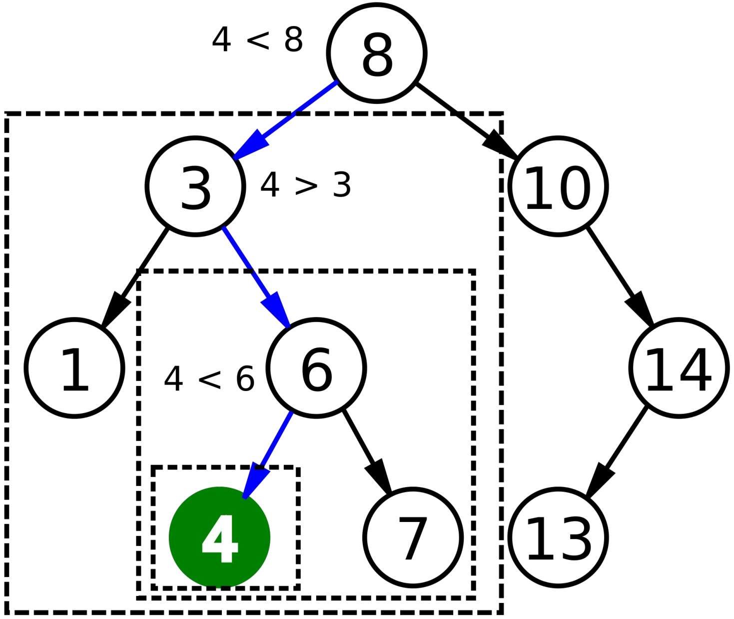


Binary Search Tree

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);
}
```



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.

```
          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

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

- 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

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

We assume the root is not `nullptr`.

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

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

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`.

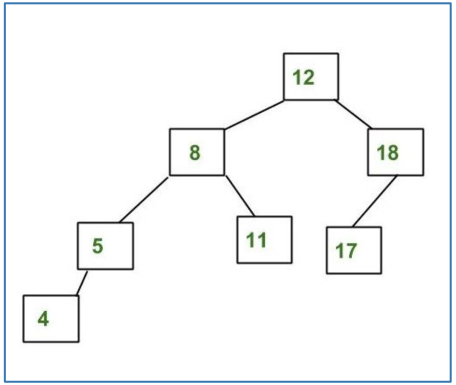
```
bool isValidBST(const node* node)
{
  if (node == nullptr) return true;
  /* check left subtree and right subtree condition*/
  if (node->left != nullptr && findMaxKey(node->left) > node->key)
    return false;
  if (node->right != nullptr && findMinKey(node->right) < node->key)
    return false;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

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

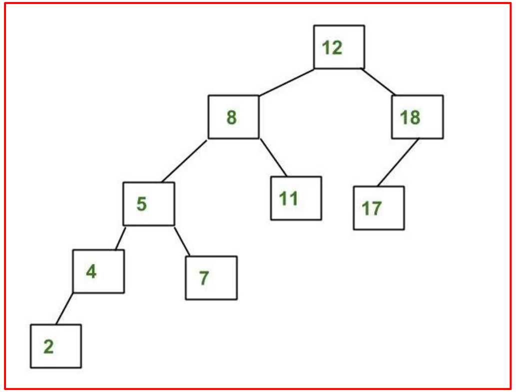
Beyond Binary Search Tree

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!



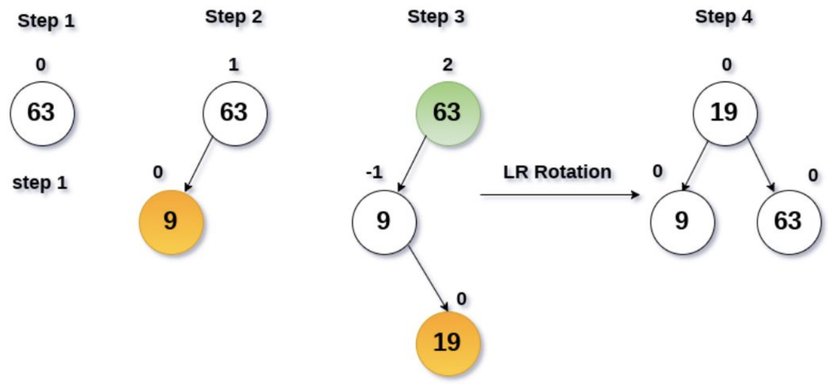
This is not OK!

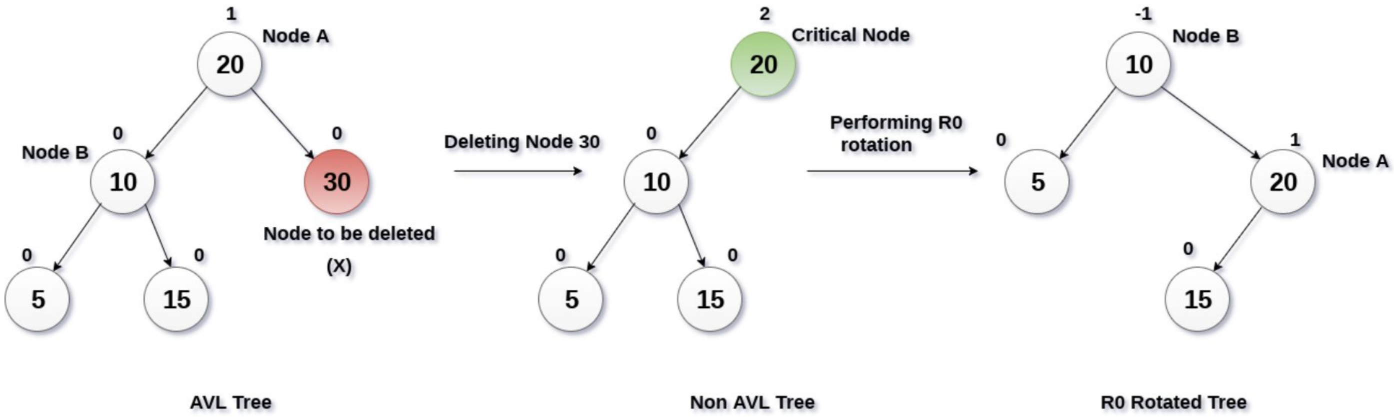
Beyond Binary Search Tree

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:

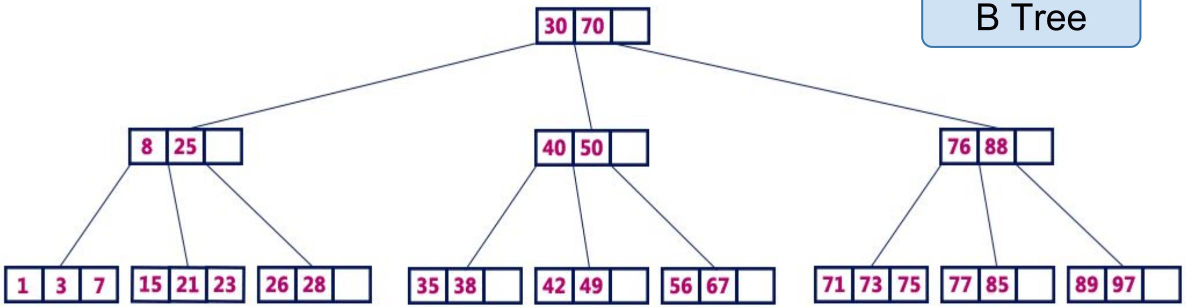
- B tree and B+ tree (I did not hear about A+ tree though)
- Quad tree
- R tree (spatial index tree)
- Red-black tree
- K-D tree

Please check the given links for more details!

Beyond Binary Search Tree

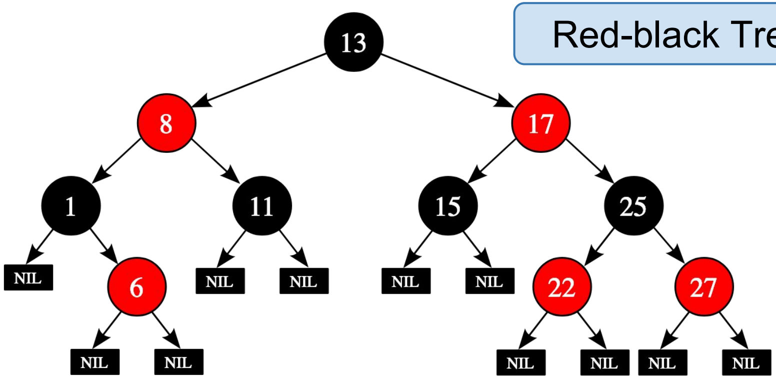
The tree family (2)

B-Tree of Order 4

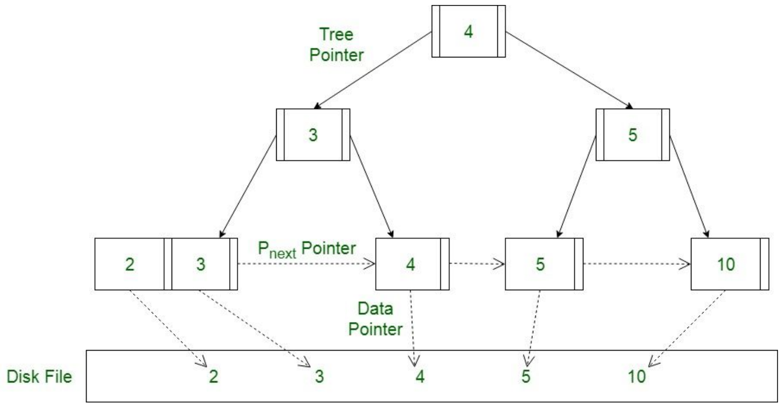


B Tree

Red-black Tree



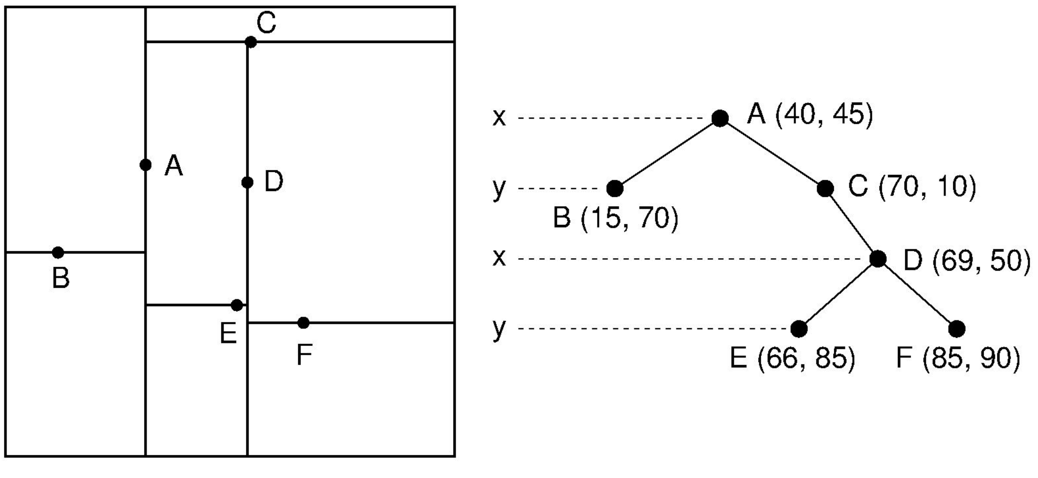
B+ Tree



Beyond Binary Search Tree

The tree family (3)

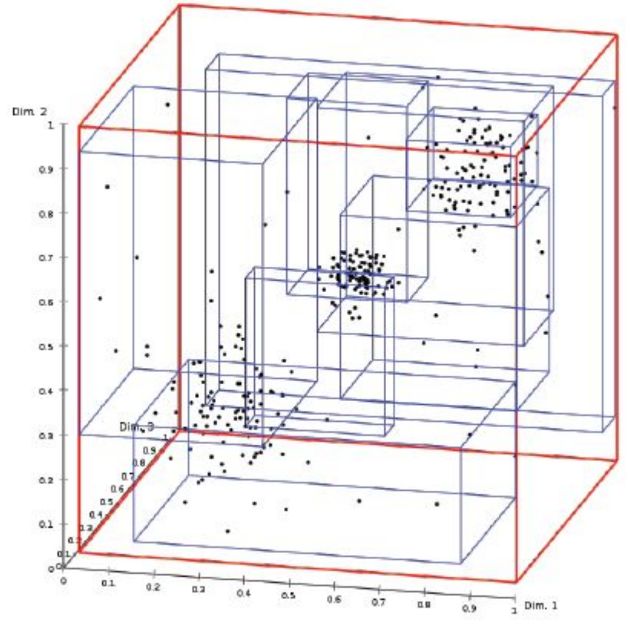
KD Tree



(a)

(b)

R Tree



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

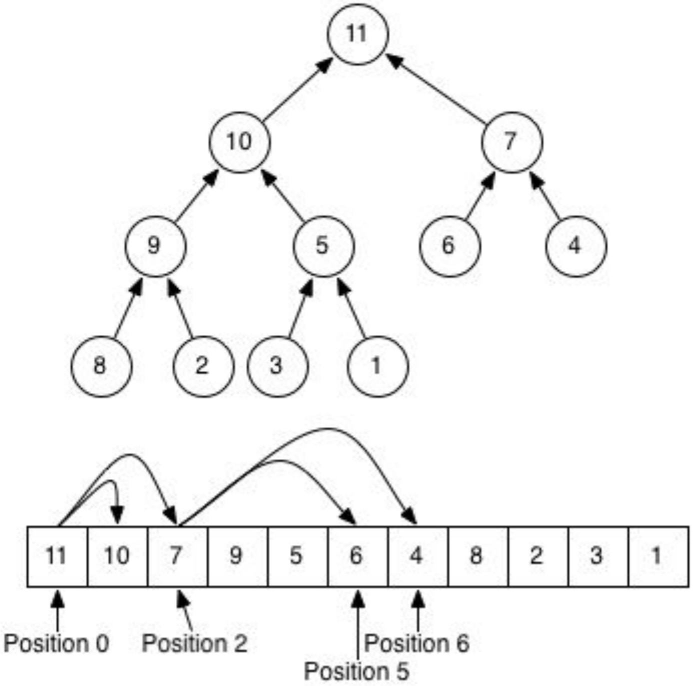
Heap

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.

Stack

Heap

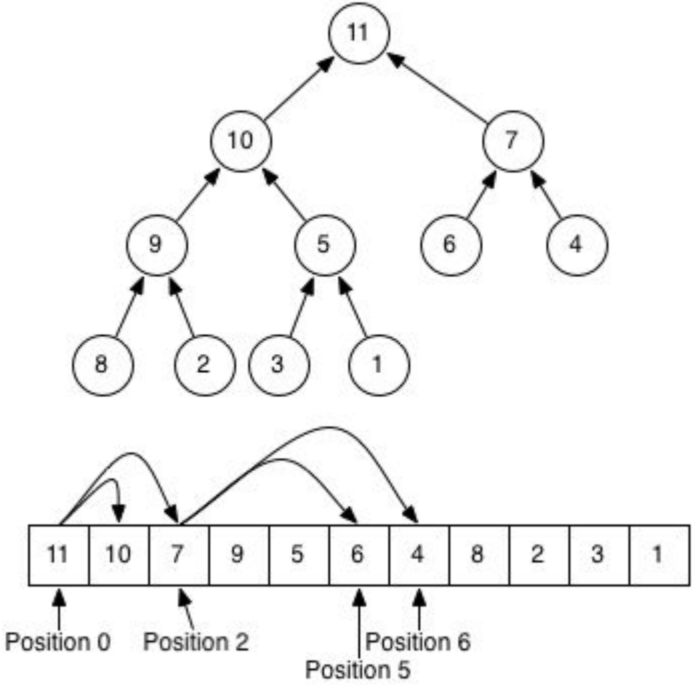


Heap

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

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

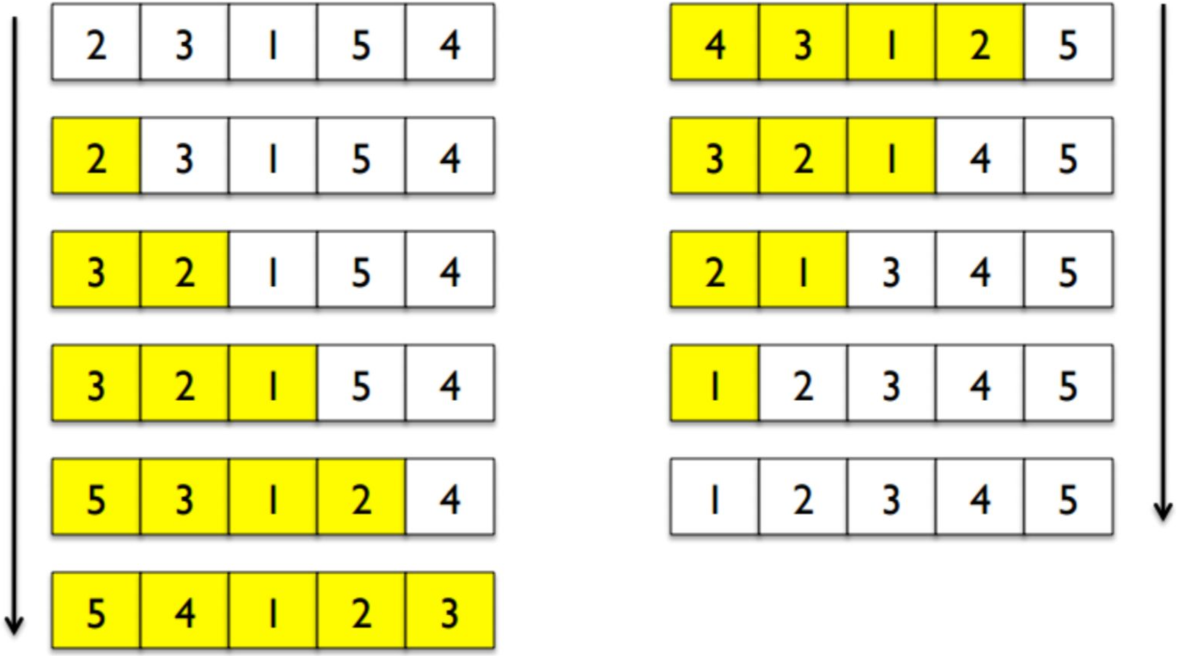
- Bonus: How can you sort based on heap?
  - Insert all elements into a heap.
  - Extract the maximum element from the heap one by one.
  - Check the example in [Wikipedia]
- What is the complexity of heapsort?
  - $O(n \log n)$

In-place Heapsort (with an array)

UCLA Samueli Computer Science

build the maxHeap

extract



part of the maxHeap

Heapsort Problem 1

Find k largest numbers

- How can we efficiently find ***k* largest numbers** from n numbers? (n>>k, but k is not small)
  - Sort? → O(*nlogn*)
  - Scan *k* times by linear search? → O(*nk*)

- Use heapsort
  - Only keep the largest
  - Whether to use MaxHeap or MinHeap?
  - Overall complexity?

Min Heap!
Pop out the `min` from heap and insert a number, if it's larger than `min`.

O(*nlogk*)

Heapsort Problem 2

Find median from a streaming data

How to find the median of the streaming data?

That is, implementing the following program:

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

Here you may use two heaps to store all the coming data → find median in O(1) time.

```
class MedianFinder {
public:
  MedianFinder() {
    // construction
  }

  void addNum(int num) {
    // add new integers from stream
  }

  double findMedian() {
    // return the median
  }
private:
    // define your private data member(s)
};
```

Heapsort Problem 3

Merge *k* linked list

- How to merge k sorted linked lists? Each list has n nodes.

- Solution 1: Brute forth $\rightarrow O(nk^2)$
  - Keep linear searching the k heads and fetching the smallest until all lists are empty

- Solution 2: Use MinHeap $\rightarrow O(nk\log k)$
  - Insert the head of each list to the heap.
  - Each time we pop-out a node from the heap and append it to the result list, insert the next node of that node from its list to the heap (until heap is empty)

- Solution 3: Merge sort $\rightarrow O(nk\log k)$
  - Merge each pair of sorted lists. k sorted lists become k/2 sorted lists.
  - Repeat the list merge from k/2 to k/4… (until everything is merged into 1 list.)

Hash Tables

Start from hashing and hash functions

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

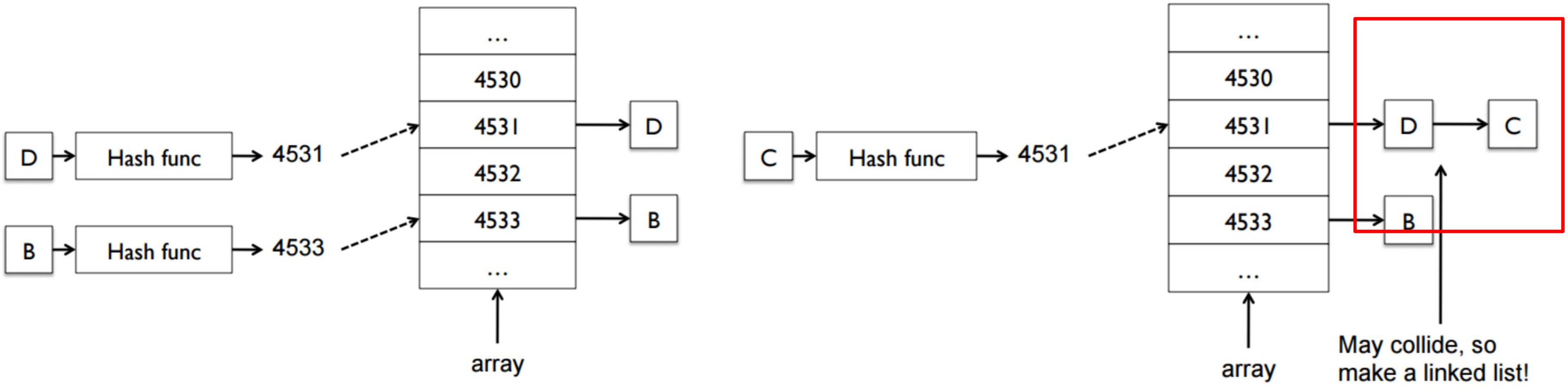

```
unsigned int FNV-1(string s) {
  unsigned int h = 2166136261U;
  for (int k = 0; k != s.size(); k++)
  {
    h += s[k];
    h *= 16777619;
  }
  return h;
}
```

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

UCLA Samueli Computer Science

- 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

The very first question in LeetCode - TwoSum

UCLA Samueli Computer Science

- 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

UCLA Samueli Computer Science

- Question: 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], target = 9. Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        /* Your solution here */
    }
};
```

UCLA Samueli Computer Science

Thank you!

Q & A

Group Exercise

Exercise problems from Worksheet 6 (see “LA worksheet” tab in CS32 website).

Questions today:

- Worksheet 6

Answers will be posted after discussions.