# Data Structure and Algorithm

## Class 7

Seongjin Lee

Updated: 2017-03-06
DSA_2017_07

insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

# Table of contents

# Tree

# Introduction

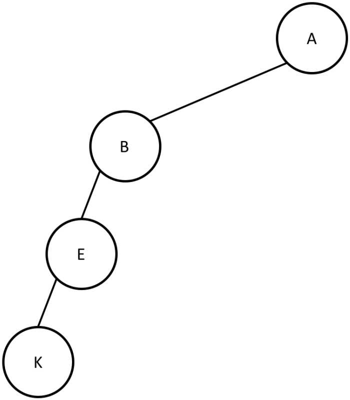
**Tree** It is finite set of one or more nodes such that

1. there is a special node called root
2. remaining nodes are partitioned into n$geq$0 disjoint trees $T_1, T_2, \cdots, T_n$ where each of these is a tree; we call each $T_i$ subtree of the root

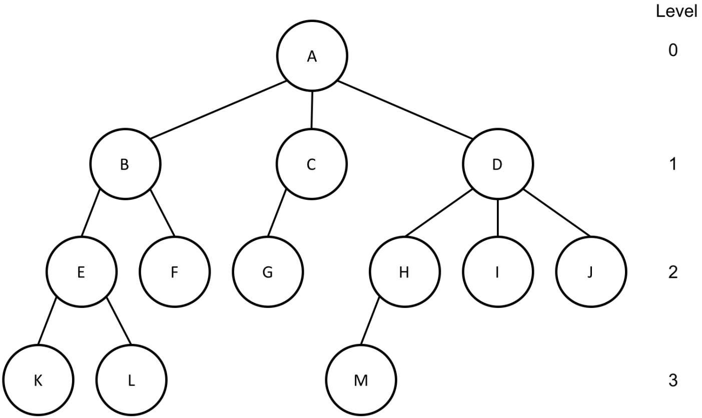
**Acyclic graph** A tree that contains no cycle

It has a hierarchical structure

# Tree



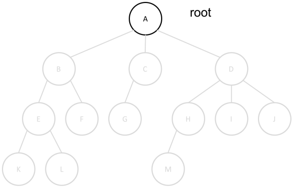
# Tree



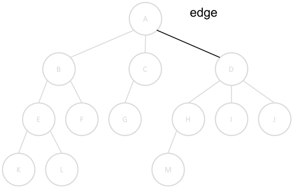
# Terminology

# Terminology I

**Root** A node with no parent (e.g., A is the root)

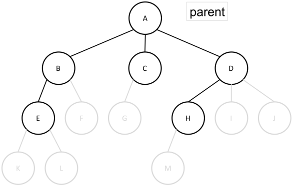



**edge** The connecting link between any two nodes (e.g., link between A and B is edge)

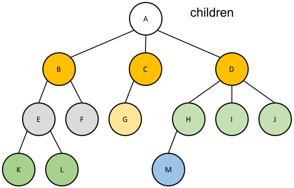


# Terminology II

**parent** a node that has subtrees (e.g., A is parent of B, C, and D. E is parent of K and L)

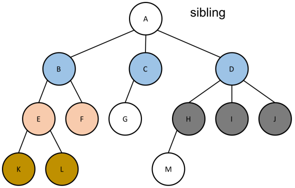



**child** a root of the subtrees (e.g., E is child of B, C is child of A)

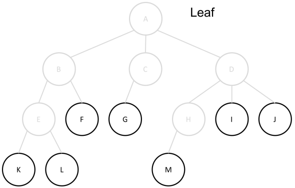


# Terminology III

**sibling** child nodes of the same parent (e.g., B, C, and D are siblings, K, L, and M are siblings)



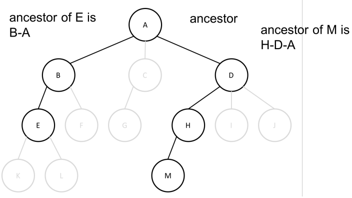

**Leaf (terminal, external) node** A node with degree zero (e.g., K, L, F, G, M, I, and J)



# Terminology IV

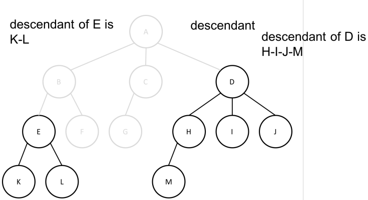
**Internal (non-terminal, internal) node** node with degree one or more (e.g., A, B, C, D, E, and H)

**ancestor** all the nodes along the path from the root to the node (e.g., ancestor of K is E, B, and A. Ancestor of H is D, and A)



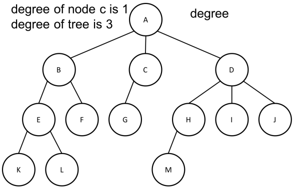
# Terminology V

**descendant** all the nodes that are in its subtrees (e.g., Descendants of E is K, and L. Descendants of D is H, M, I, and J)

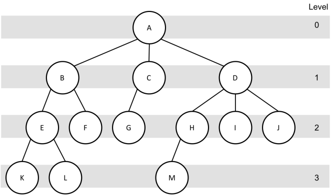



**Degree of a node** The number of subtrees of node, in other words the total number of children of a node (e.g., Degree of A is 3. Degree of C is 1)

**Degree of a tree** The maximum degree of the nodes in the tree (e.g., the degree of the tree is 3)



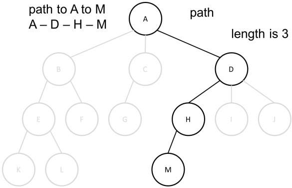

**level** each step from top to bottom, the level of the root node is 0 (e.g., level of F is 2. Level of M is 3)



# Terminology VII

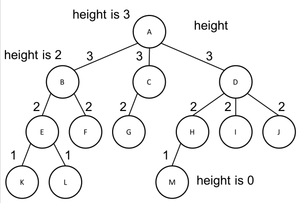
**path** the set of edges from the root to a node (e.g., the path to M from A is (A, D), (D, H), (H, M))

**path length** The number of edges in a path (e.g., path length from A to M is 3)



# Terminology VIII

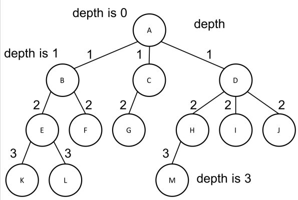
**Height of a tree** The longest path length from the root to a leaf (e.g., the height is 3)



# Terminology IX

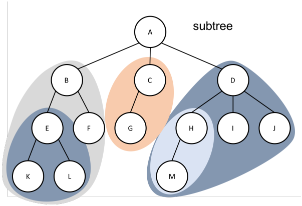
**depth** the total number of edges from root node to a particular node (e.g., depth of F is 2. Depth of M is 3)

**depth of the tree** the total number of edges from root node to a leaf node in the longest path (e.g., depth of the tree is 3)

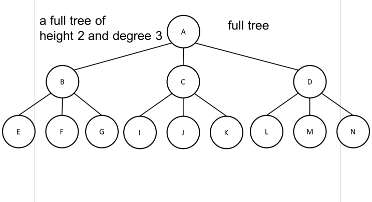


# Terminology X

**subtree** each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node

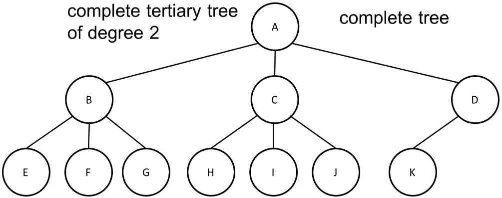



**proper (or full) tree** every node other than the leaves has non-void children



# Terminology XI

**complete tree** All levels are full except for the deepest level, which is partially filled from the left



# Tree Representation

# Tree Data Structure

Two types of representation of tree

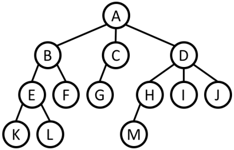
1. List representation
2. left child - right sibling representation

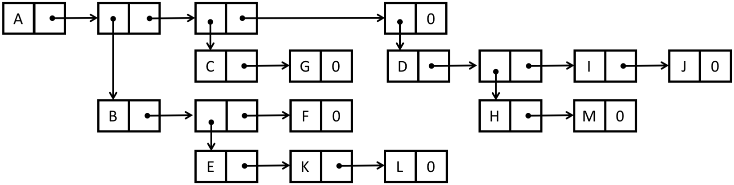
# List Representation

Two types of nodes

1. Node with data
2. Node with the reference

The information in the root node comes first and it is followed by a list of the subtrees of that node

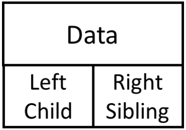


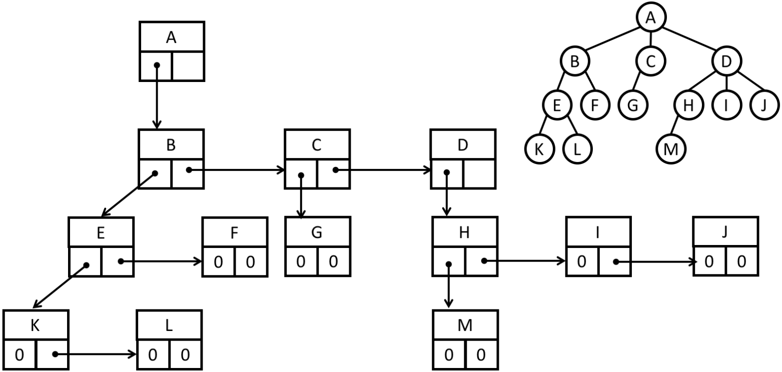
# Left-child Right-Sibling Representation

nodes of a fixed size

- easier to work
- two link / pointer fields and a data field per node
  - left reference field stores the address of the left child
  - right reference field stores the address of the right sibling node



# Left-child Right-sibling Representation

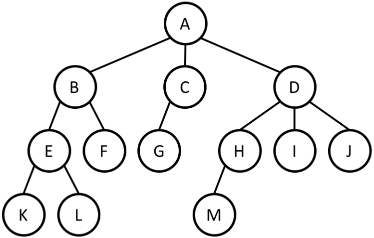


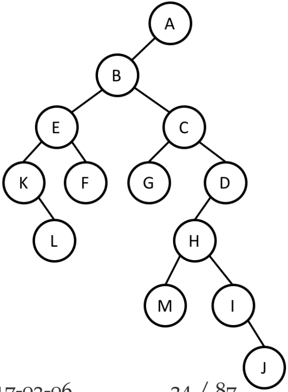
# Representation of trees

Changing a representation of a tree as a degree two tree

- simply rotate the left-child right-sibling tree clockwise by 45 degrees

A tree with degree 2 (two children, left and right child) is called a binary tree



# Binary Tree

# Definition

A binary Tree is a finite set of nodes such that

1. empty or
2. consists of root note and two disjoint binary trees, called left subtree and right subtree

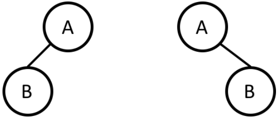



**Figure:** two different types of binary tree

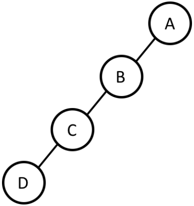
# Properties

Difference between a binary tree and a tree

- may have empty node
- the order of subtree are important
- a binary tree is not a subset of a tree
- maximum number of nodes in a Binary Tree is $2^k - 1$ where k is depth of the tree
- relationship between the number of leaf nodes ($n0$) and the number of nodes with degree 2 ($n2$) $n0 = n2 + 1$

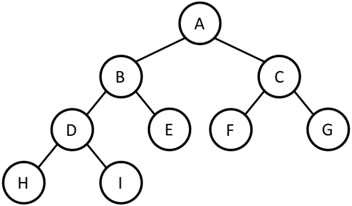
# Special Types of Binary Trees I

- skewed binary tree



- full binary tree (of depth k)

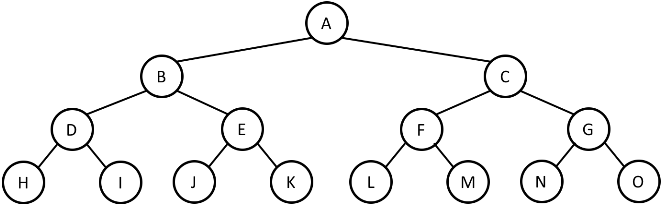
  **full (or proper or strictly binary tree** every node has either two or zero number of children



# Special Types of Binary Trees III

- complete binary tree

**complete (or perfect) binary tree** A binary tree in which every internal node has exactly two children and all leaf nodes are at same level a binary tree with n nodes that correspond to the nodes numbered from 1 to n in the full binary tree of depth k



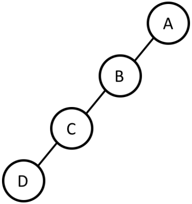
# Binary Tree Representation

There are two methods to represent the binary tree

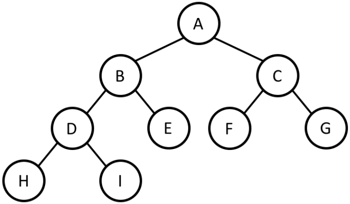
1. Array Representation
2. Linked List Representation

# Array Representation

- sequential representations
- determine the locations of the parent, left child, and right child of any node i in the binary tree
  1. parent (i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$, if $i = 1$, no parent
  2. left_child (i) is at $2 \cdot i$ if $2i \leq n$
  3. right_child (i) is at $2 \cdot i + 1$ if $2 \cdot i + 1 \leq n$




| | |
|---|---|
| 1 | A |
| 2 | B |
| 3 | - |
| 4 | C |
| 5 | - |
| 6 | - |
| 7 | - |
| 8 | D |
| 9 | - |
| 10 | - |
| 11 | - |
| 12 | - |
| 13 | - |
| 14 | - |
| 15 | - |
| 16 | E |




| | |
|---|---|
| 1 | A |
| 2 | B |
| 3 | C |
| 4 | D |
| 5 | E |
| 6 | F |
| 7 | G |
| 8 | H |
| 9 | I |

# The Problem of Array Representation of Tree

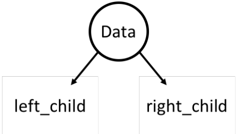
- inefficient storage utilization
  $S(n) = 2 - 1$ where k is depth of binary tree
  ideal for complete binary trees
- hard to insert/delete

# Linked List representation

## Representing tree with linked list

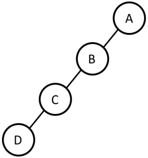
- each node has three fields
  1. left_child
  2. data
  3. right_child

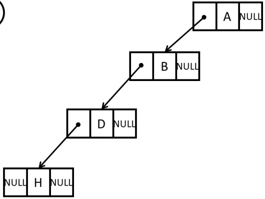
```c
typedef struct BinaryTreeNode {
    int data;
    struct BinaryTreeNode* left_child;
    struct BinaryTreeNode* right_child;
} node;
```

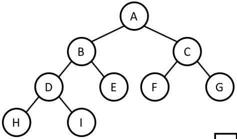
# example

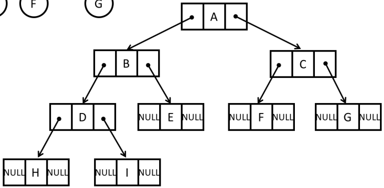
Skewed




complete

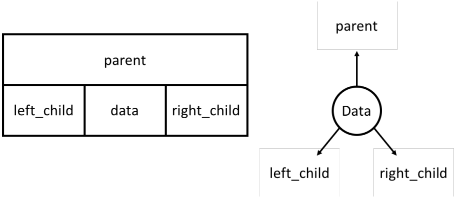


# Linked List Representation cont'd

leaf node's link field contains NULL pointer


Add a fourth field, called parent, to know the parent of a random nodes



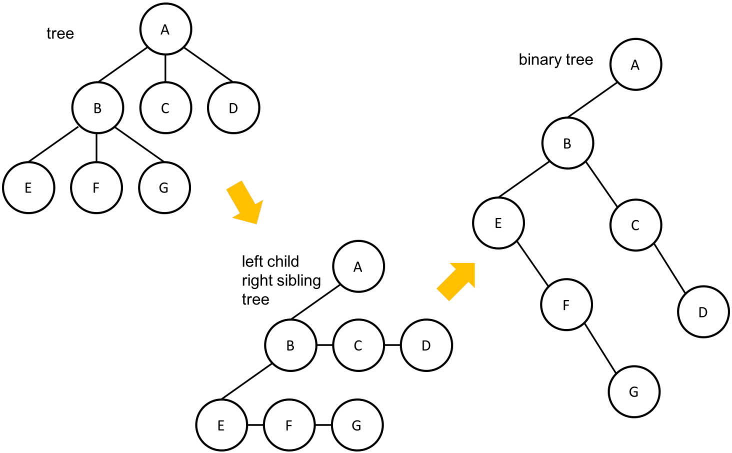
# Tree Representation

- each node in a tree has a variable sized nodes
- hard to represent it by using array
- use linked list to represent a tree needs $k$ link fields per node
  - $k$ is the degree of tree
- There are two types of links
  - non-null links
  - null links
- if the number of non-null links are $n - 1$
  - the number of null links are $n \cdot k - (n - 1)$

# Converting a tree into a binary Tree

1. Use left-child right sibling representation
   - (parent, $child_1$, $child_2$, ..., $child_x$) $\rightarrow$ (parent, leftmost-child, next-right-sibling)
2. simply rotate the left-child right-sibling tree clockwise by 45 degrees
   - right field of root node always have null link
   - null links: approximately 50%
   - depth increased

# Converting a tree into a binary Tree



# Binary Tree Traversal and Other Operations

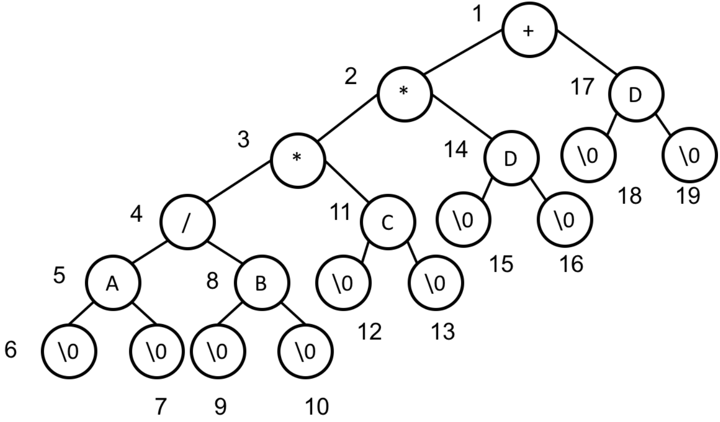
# Binary Tree Traversals

visit each node in the tree exactly once

- produce a linear order for the information in a tree
- what order?
  - inorder: LVR (Left Visit Right)
  - preorder: VLR (Visit Left Right)
  - postorder: LRV (Left Right Visit)

# Binary Tree Traversals

$A / B * C * D + E$ (infix form)



# Binary Tree Traversals

## Inorder Traversal

```
void inorder(TreeNode *ptr) {
    if(ptr) {
        inorder(ptr->left_child);
        printf("%d",ptr->data);
        inorder(ptr->right_child);
    }
}
```

inorder is invoked 19 times for the complete traversal: 19 nodes

output: $A / B * C * D + E$

- corresponds to the infix form

# Binary Tree Traversal

| call of in-order | value in root | action |
|---|---|---|
| 1 | + | |
| 2 | * | |
| 3 | * | |
| 4 | / | |
| 5 | A | |
| 6 | NULL | |
| 5 | A | printf |
| 7 | NULL | |
| 4 | / | printf |
| 8 | B | |
| 9 | NULL | |
| 8 | B | printf |
| 10 | NULL | |
| 3 | * | printf |

| call of in-order | value in root | action |
|---|---|---|
| 11 | C | |
| 12 | NULL | |
| 11 | C | printf |
| 13 | NULL | |
| 2 | * | printf |
| 14 | D | |
| 15 | NULL | |
| 14 | D | printf |
| 16 | NULL | |
| 1 | + | printf |
| 17 | E | |
| 18 | NULL | |
| 17 | E | printf |
| 19 | NULL | |

# Preorder Traversal

```c
void preorder(TreeNode *ptr) {
    if(ptr) {
        printf("%d",ptr->data);
        preorder(ptr->left_child);
        preorder(ptr->right_child);
    }
}
```

output in the order $+ * * / ABCDE$

# Postorder Traversal

```
void postorder(TreeNode *ptr) {
    if(ptr) {
        postorder(ptr->left_child);
        postorder(ptr->right_child);
        printf("%d",ptr->data);
    }
}
```

output in the order $AB/C*D*E+$

# Iterative Inorder Traversal

Recursion

- call itself directly or indirectly
- simple, compact expression: good readability
- don't need to know implementation details
- much storage: multiple activations exist internally
- slow execution speed
- application: factorial, Fibonacci number, tree traversal, binary search, tower of Hanoi, quick sort, LISP structure

# Iterative Inorder traversal

```c
void iter_inorder(tree_ptr node) {
    int top = -1;
    tree_ptr stack[MAX_STACK_SIZE];
    while (1) {
        while (node) {
            push(&top, node);
            node = node->left_child;
        }
        node = pop(&top);
        if (!node) break;
        printf("%d", node->data);
        node = node->right_child;
    }
}
```

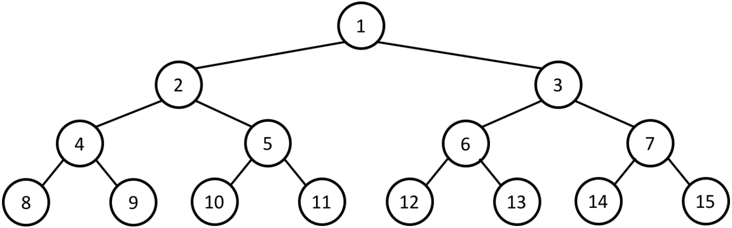
# Iterative Inorder Traversal

every node of the tree is placed on and removed from the stack exactly once

- time complexity: O(n) where n is the number of nodes in the tree
- space complexity: stack size O(n) where n is worst case depth of the tree (case of skewed binary tree)

# Level Order Traversal

Traversal by using queue (FIFO)




Output in the order: 1, 2, 3, 4, . . ., 14, 15

# Level Order Traversal

```c
void level_order(tree_ptr ptr) {
    int front = rear = 0;
    tree_ptr queue[MAX_QUEUE_SIZE];

    if (!ptr) return;

    addq(front,&rear,ptr);

    while (1) {
        ptr = deleteq(&front, rear);

        if (ptr) {
            printf("%d", ptr->data);
            if (ptr->left_child)
                addq(front,&rear,ptr->left_child);
            if (ptr->right_child)
                addq(front,&rear,ptr->right_child);
            else break;
        }
    }
}
```

# Copying Binary Tree

Modified version of postorder

```
tree_ptr copy(tree_ptr original) {
    tree_ptr temp;
    if (original) {
        temp = (tree_ptr)malloc(sizeof(node));

        if (IS_FULL(temp))
            exit(1);
        temp->left_child = copy(original->left_child);
        temp->right_child = copy(original->right_child);
        temp->data = original->data;
        return temp;
    }
    return NULL;
}
```

# Testing for equality of binary trees

## Modified version of preorder

```
int equal(tree_ptr first, tree_ptr second) {
    return ((!first && !second) ||
             (first && second
                   && (first->data == second->data)
                   && equal(first->left_child, second->left_child)
                   && equal(first->right_child, second->right_child));
}
```

# Heaps

# Heaps: Definition

**MAX (or MIN) Tree** a tree in which the key value in each node is no smaller (larger) than the key value in its children (if any)

**MAX (or MIN) Heap** a complete binary tree that is also a max (or min) tree

- the root of a max (or min) tree contains the largest (smallest) key in the tree

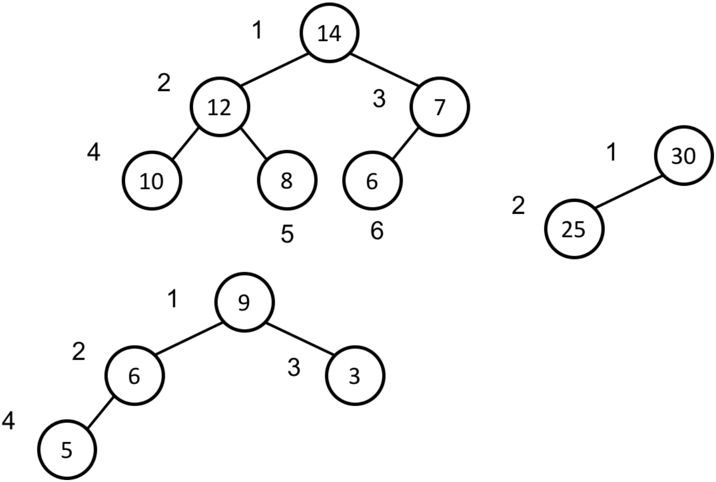
# Representation of MAX (or MIN) heaps

- array representation because heap is a complete binary tree
- simple addressing scheme for parent, left(right) child

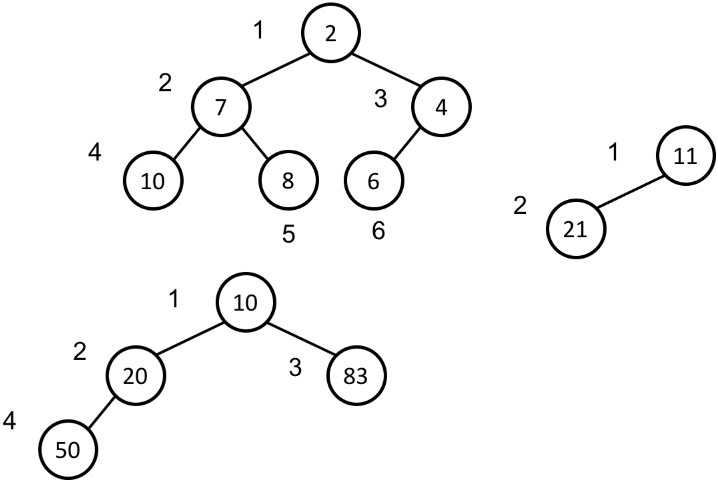
# Heap Structure

```
#define MAX_ELEMENT 200

typedef struct {
   int key; // may include other fields
} element;

typedef struct {
   element heap[MAX_ELEMENT];
   int heap_size;
} HeapType;

void init(HeapType *h){
   h->heap_size = 0;
}
```

# Sample Max Heaps



# Sample Min Heaps



# Priority Queues

**deletion** deletes the element with the highest(or the lowest) priority

**insertion** insert an element with arbitrary priority into a priority queue at any time

Ex. Job scheduling of OS

# Priority Queues

We use a max (or Min) Heap to implement the Priority Queues

Possible priority queue representations

| Representation | insertion | deletion |
|---|---|---|
| unordered array | O(1) | O(n) |
| unordered linked list | O(1) | O(n) |
| sorted array | O(n) | O(1) |
| sorted linked list | O(n) | O(1) |
| max heap | O($log_2 n$) | O($log_2 n$) |

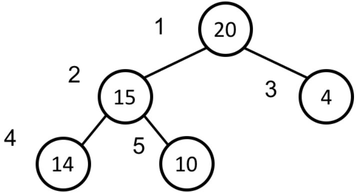
# Insertion into a max heap

Need to go from a node to its parent

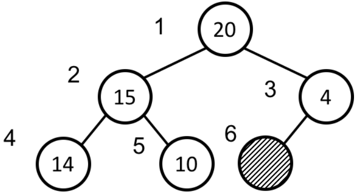
- linked representation add a parent field to each node
- array representation a heap is a complete binary tree simple addressing scheme

# Insertion into a max heap

Heap before Insertion

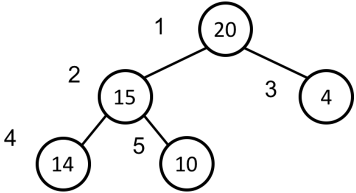



Initial location of new node

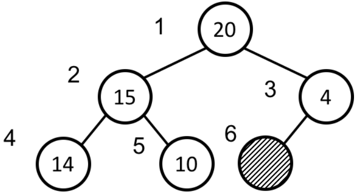


# Insertion into a max heap

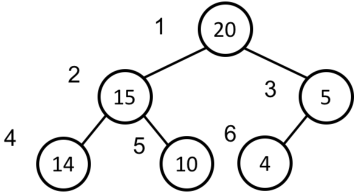
Heap before Insertion



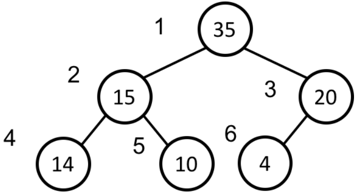

Initial location of new node




Insert 5 into heap




Insert 35 into heap



# Insertion into a max heap

- select the initial location for the node to be inserted $\rightarrow$ bottommost-rightmost leaf node
- insert a new key value adjust key value from leaf to root parent position: $\lfloor i/2 \rfloor$
- time complexity : $O(depthoftree) \rightarrow O(log_2 n)$

# Insertion into a max heap

Refer to page 57 for definitions and sturctures

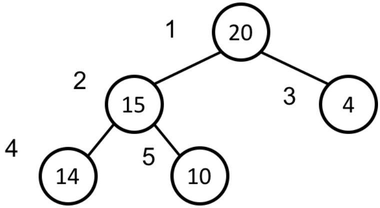
```c
// add item to the heap
void insert_max_heap(HeapType *h, element item){
    int i;
    i = ++(h->heap_size); // increase the heap

    // traverse to the top of the max heap tree
    // if item is lareger than the parent's item (heap[i/2].key)
    while((i != 1) && (item.key > h->heap[i/2].key)){
        h->heap[i] = h->heap[i/2];
        i /= 2;
    }
    h->heap[i] = item; // add new node to the heap
}
```

# Deletion from a max heap

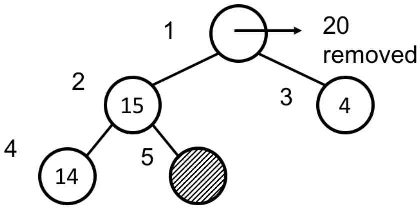
- always delete an element from the root of the heap
- restructure the tree so that it corresponds to a complete binary tree
- place the last node to the root and from the root compare the parent node with its children and exchanging out-of-order elements until the heap is reestablished

# Deletion from a max heap

Heap before deletion

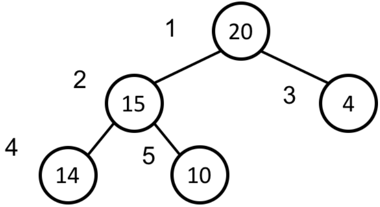



Heap Structure

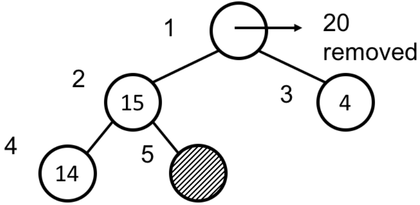


# Deletion from a max heap

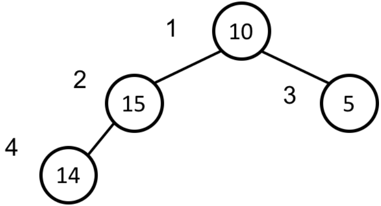
Heap before deletion



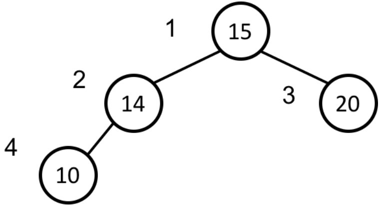

Heap Structure




10 inserted at the root




Final heap



# Deletion from a max heap

- select the removed node bottommost-rightmost leaf node
- place the node's element in the root node
- adjust key value from root to leaf compare the parent node with its children and exchange out-of-order elements until the heap is reestablished -
- time complexity : $O(depth\ of\ tree) \rightarrow\ O(log_2 n)$

# Deletion from a max heap I

Refer to page 57 for definitions and sturctures

```
// delete the item
element delete_max_heap(HeapType *h){
    int parent, child;
    element item, temp;

    item = h->heap[1]; // take the max value from the heap
    temp = h->heap[(h->heap_size)--]; // reduce the heap size

    // initial position
    parent = 1;
    child = 2;

    while( child <= h->heap_size ){ // within the heap
        // loop until counted number of child is less the the heap size
        // find the larger key in the heap
        if( ( child < h->heap_size ) &&
            ( h->heap[child].key ) < h->heap[child+1].key )
            child++;

        // if found key is smaller than the last key in the tree
        // take the last key
```

# Deletion from a max heap II

```
        if( temp.key >= h->heap[child].key )
            break;

        // take the child and advance
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2; // increase the position
    }

    h->heap[parent] = temp;
    return item; // return the deleted value
}
```

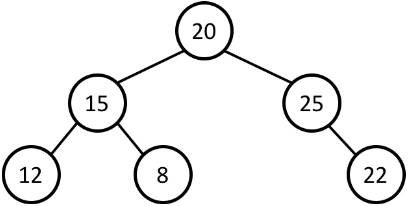
# Binary Search Tree

# Binary Search Tree (BST)

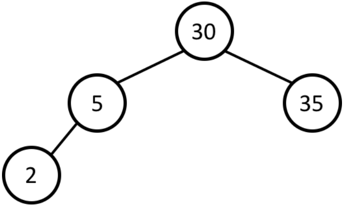
Binary search tree(BST) is a binary tree that is empty or each node satisfies the following properties:

1. every element has a key, and no two elements have the same key
2. the keys in a nonempty left subtree must be smaller than the key in the root of the subtree
3. the keys in a nonempty right subtree must be larger than the key in the root of the subtree
4. the left and right subtrees are also BST

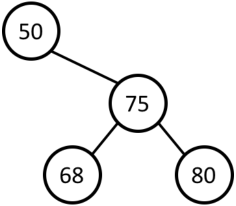
Not a BST




BST




BST



# Operations and their Complexity

Searching, Insertion, Deletion is bounded by $O(h)$ where h is the height of the BST

can perform these operations both

- by key value and
  e.g., delete the element with key x
- by rank
  e.g., delete the fifth smallest element

## Recursive search of a BST

```
tree_ptr search(tree_ptr root, int key) {
    /* return a pointer to the node that contains
     * key. If there is no such node, return NULL
     */
    if (!root) return NULL;
    if (key == root->data) return root;
    if (key < root->data)
       return search(root->left_child, key);
    return search(root->right_child, key);
}
```

# Iterative Search of a BST

```
tree_ptr iter_search(tree_ptr tree, int key) {
    while (tree) {
        if (key == tree->data) return tree;
        if (key < tree->data)
            tree = tree->left_child;
        else
            tree = tree->right_child;
    }
    return NULL;
}
```

# Time complexity for searching

- Average case
  - $O(h)$ where h is the height of BST
- Worst case
  - $O(n)$ for skewed binary tree

# Inserting into a BST I

```
#include <stdio.h>
#include <stdlib.h>

// tree node data structure
typedef struct node *tree_ptr;
typedef struct node {
   int data;
   tree_ptr left_child, right_child;
} node;


// modification of recursive search
// returns the node
tree_ptr compare(tree_ptr root, int key){
    if (!root) return NULL;
    if (key < root->data){
       root->left_child = compare(root->left_child, key);
       return root;
    } else {
       root->right_child = compare(root->right_child, key);
       return root;
    }
    return root;
```

# Inserting into a BST II

```
}

// insert the new value
void insert_node(tree_ptr *node, int num) {
    tree_ptr ptr;
    tree_ptr temp;
    temp = compare(*node, num);
    if (temp || !(*node)) {
        ptr = (tree_ptr)malloc(sizeof(node));
        //if (IS_FULL(ptr)) {
         // fprintf(stderr,"The memory is full\n");
          // exit(1);
        //}
        ptr->data = num;
        ptr->left_child = ptr->right_child = NULL;
        if (*node)
            if (num < temp->data)
                temp->left_child = ptr;
            else
                temp->right_child = ptr;
        else *node=ptr;
    }
}
```

# Inserting into a BST III

```
int main(){
    tree_ptr *new;
    insert_node(new, 3);
    insert_node(new, 4);
    insert_node(new, 5);
    return 0;
}
```

# Inserting into a BST

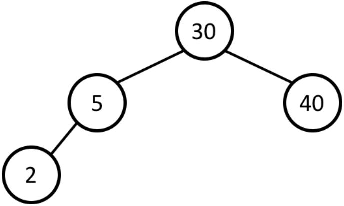
- return NULL, if the tree is empty or num is present
- otherwise, return a pointer to the last node of the tree that was encountered during the search

time complexity for inserting

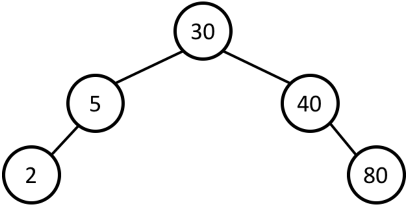
- $O(h)$, where h is the height of the tree

# Inserting into a BST

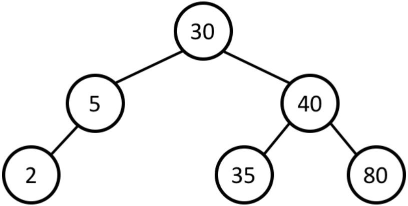
Initial BST




Insert 80




Insert 35



# Deleting from BST

deletion of a leaf node

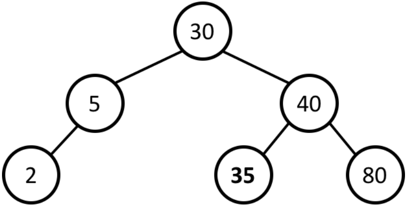
- deletion of a node with 1 child
- deletion of a node with 2 children

time complexity for deleting

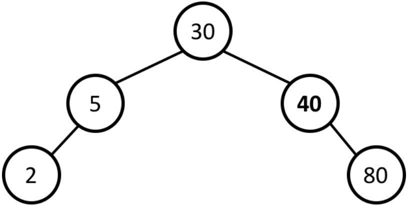
- $O(h)$ where h is the height of the tree

# Deleting a leaf or a node with a child

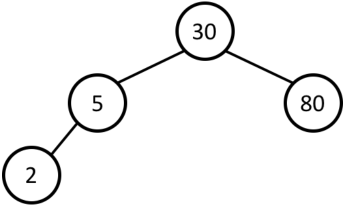
Initial BST




Delete 35 (leaf)

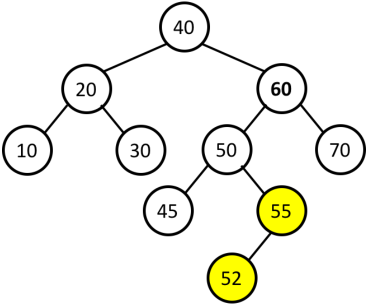



Delete 40 (node with single child)

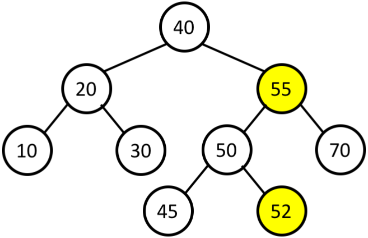


# Deletion of a node with two children

Tree before deletion of 60




Tree after deletion of 60



# Height of a BST

the height of a BST with $n$ elements

- average case: $O(log_2 n)$
- worst case: $O(n)$
  - e.g., use `insert_node` to insert the keys $1, 2, 3, \ldots, n$ into an initially empty BST

# Balanced (binary) Search Tree

- worst case height: $O(log_2 n)$
- searching, insertion, deletion is bounded by $O(h)$ where h is the height of a binary tree
- *AVL tree, 2-3 tree, red-black tree are all introduced in Chapter 10*