Data Structures
Lec-6 Trees and Heaps, and Reviews

Midterm Schedule

Written-Based

- **Wednesday Mar 6, 2024**, Lecture Time
- 9:10 – 11:10(2 Hrs)
- the content will be up to and include everything in the first 6 weeks;
- in our normal lecture room LT-6;
- **Format (total 16 Qs):**
  - Answer whether the following statements are correct or not.
    - *Time complexity of Insertion in XXX is O(n)*.
  - What is the output of the following program?
  - What is the running time of the following code?
  - Show the result of doing operation xxx.
  - Complete the implementation so that the program can XXX.

Midterm Schedule

Programming

- **Wednesday Mar 13, 2024**, Lecture Time
- Using Online Judge system.
- 9:10 – 11:10(2 Hrs), **A total of 4 questions** will be provided
  - 1 correctly-answered Q will get 40pts, 2 correctly-answered Qs will get 70pts, >= 3 correctly-answered Qs will get 100pts.
  - Submit and try the best you can as partial credits will be considered **when 0 Qs is correctly answered**
- CSC computer rooms in AC2 and CS Lab in MMW
- Content will **be up to and include basic *Tree (week 5).***
- **Do not share/copy code** as we will be running checking-software
- Checking online library documents is allowed
- Feel free to use your own laptop, draft paper, etc, but you **cannot discuss with your classmate(s)**

Midterm Schedule

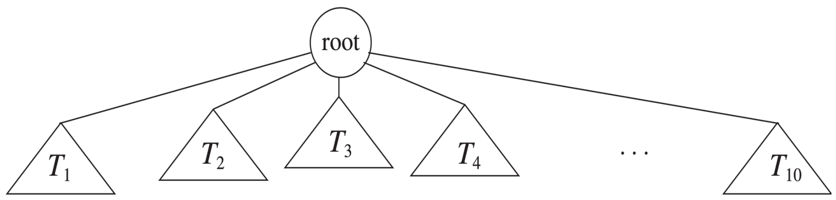
Programming

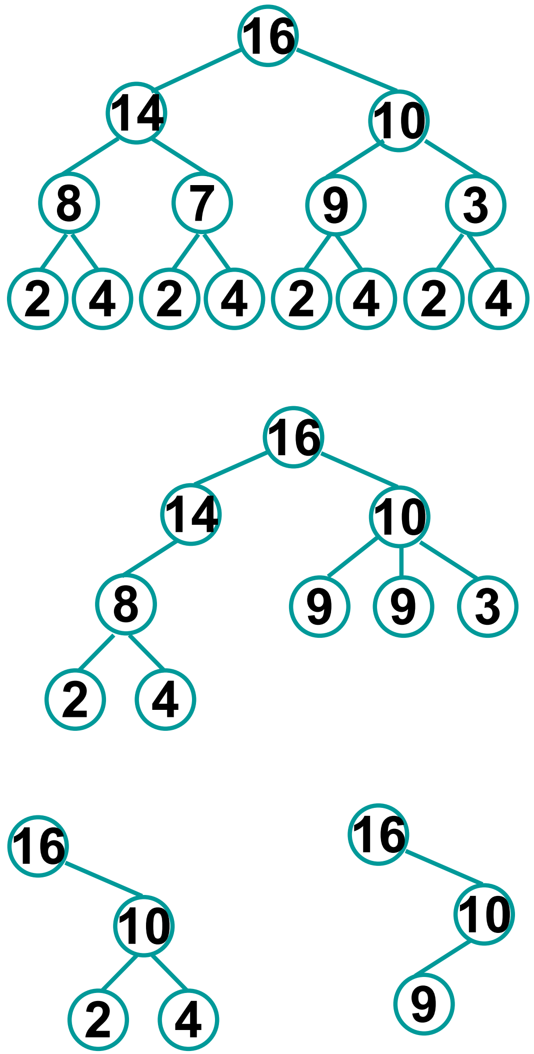
- Using Remote Desktop Service (if can't not connect to the OJ system)
  - **https://cslab.cs.cityu.edu.hk/services/cslab-campus-wide-remote-desktop-service**
  - **https://cslab.cs.cityu.edu.hk/services/macos-remote-desktop-service**
- **Do not share/copy code** as we will be running checking-software

Objective

- Definition and Terminology
- Binary Tree
  - Operations
  - Recursive functions
  - Traversal
- Binary Search Tree
  - Insertion, **deletion**
- Binary Representation of General Tree

Definition and Terminology





4 examples

Tree is defined as a finite set *T* of one or more nodes such that:

- a) there is one specially designated node called **the root of the tree**, *root (**T**)* and
- b) the remaining nodes (excluding the root) are partitioned into *m* disjoint sets $\{T_1, T_2, \ldots, T_m\}$ and each of these sets in turn is a tree. The trees $\{T_1, T_2, \ldots, T_m\}$ are called **the subtrees of the root**.

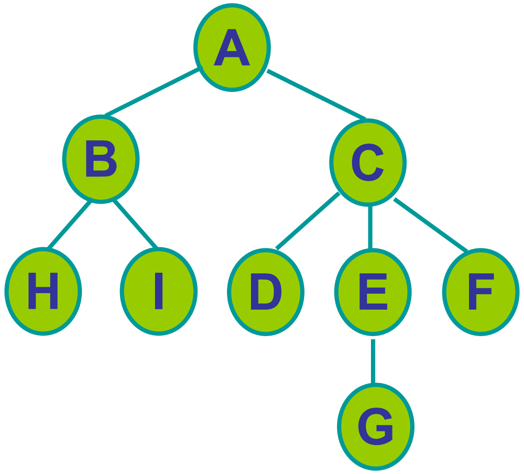
For subtrees $T_1, T_2, \ldots, T_m$, each of their roots are connected by a directed edge from the root node.

Definition and Terminology

Terminology:

Degree of a node	The number of subtrees of a node
Terminal node or **leaf**	A node of degree zero
Branch node or **internal node**	A nonterminal node
Parent and **Siblings**	Each node is said to be the parent of the roots of its subtrees, and the latter are said to be siblings; they are children of their parent.
A Path from n_1 to n_k	a sequence of nodes $n_1, n_2, \ldots n_k$ such that n_i is the parent of n_{i+1} for $0<i<k$. The length of this path is the number of edges on the path
Ancestor and **Descendant**	If there is a path from n_1 to n_k, we say n_k is the descendant of n_1 and n_1 is the ancestor of n_k .
Level or **Depth** of node	The length of the unique path from root to this node.
Height of a tree	The maximum level of any leaf in the tree.

Definition and Terminology



Level of node : State the levels of all the nodes:

A:_____, B:_____, C:_____,
D:_____, E:_____, F:_____,
G:_____, H:_____, I:_____

Root of a tree: Root of the tree is: ______

Height of a tree: Height of the tree is: ______

Degree of a node : State the degrees of:

A:_____, B:_____, C:_____,
D:_____, E:_____, F:_____,
G:_____, H:_____, I:_____

Terminal node or **leaf**: State all the leaf nodes: ____________________________

Branch node: State all the branch nodes: ____________________________

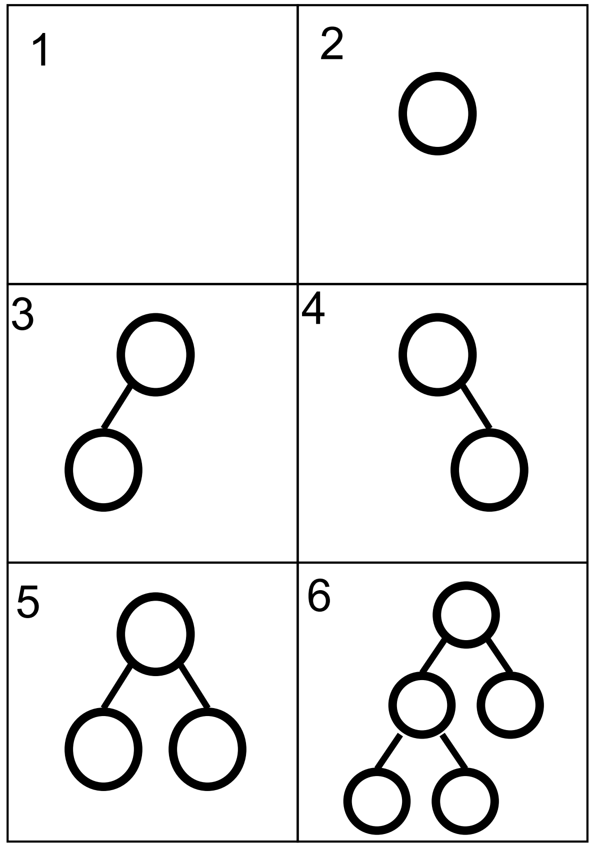
Binary Tree

Definition:

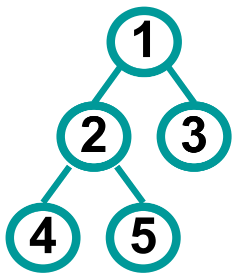
Binary tree can be defined as a finite set of nodes that either

- is empty, or
- consists of
  - (1) a root, and
  - (2) the elements of 2 disjoint binary trees called the left and right subtrees of the root.

6 Examples of Binary tree:



Properties of Binary Tree

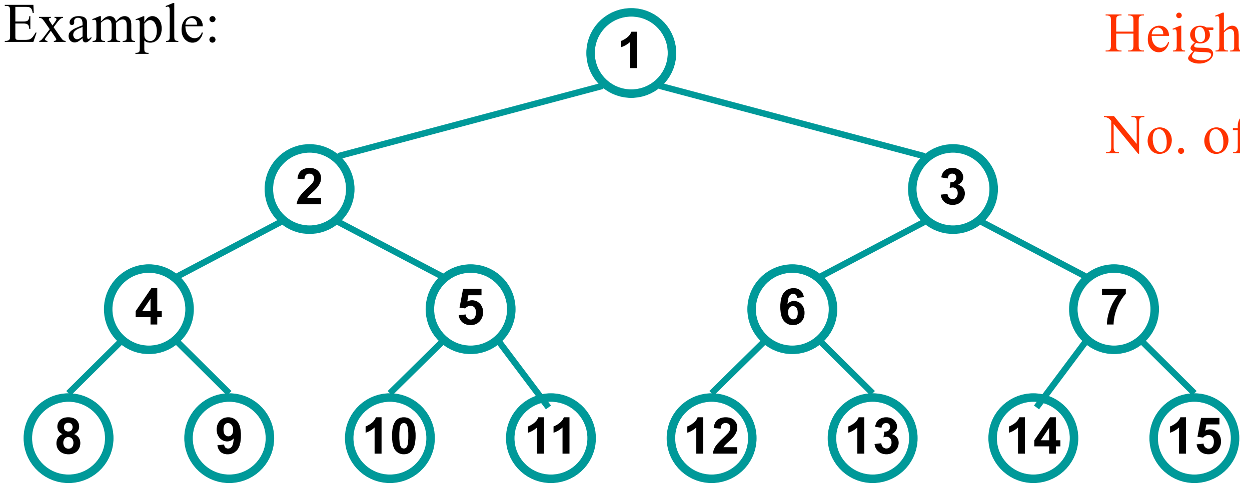


Maximum number of nodes

- Consider the levels of a binary tree: level 0, level 1, level 2, ..
- Maximum number of nodes on a level is $\underline{2^{level\_id}}$.
- Maximum number of nodes in a binary tree is $\underline{2^{height\_of\_tree+1} - 1}$.

Full Binary Tree: **No. of nodes = $2^{height\_of\_tree +1}$-1**

Example:



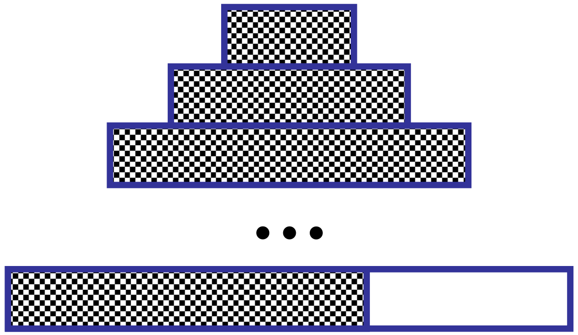
Height of tree = 3

No. of nodes $= 2^{height\_of\_tree +1}-1$

$= 15$

Properties of Binary Tree

Complete Binary Tree

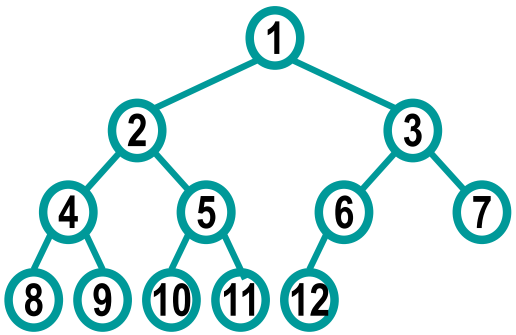


A complete binary tree is like a full binary tree,
But in a complete binary tree,

- Except the bottom level: all are fully filled.
- The bottom level: The filled slots are at the left of the empty slots (if any).

Definition: A binary tree with n nodes and height k is **complete** if and only if its nodes correspond to the nodes numbered from 1 to n in the fully binary tree of height k.

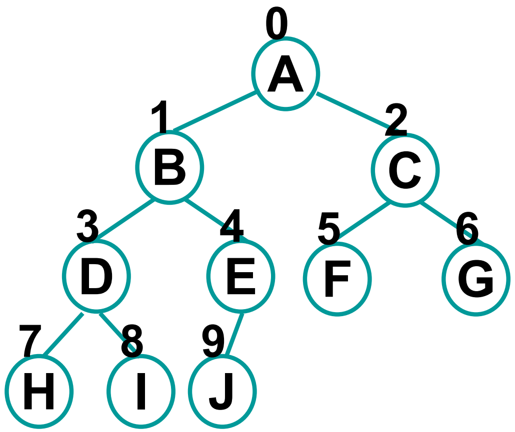
- Each leaf in a tree is either at level k or level k-1
- Each node has exactly 2 subtrees at level 0 to level k-2



Array Representation of Binary Tree

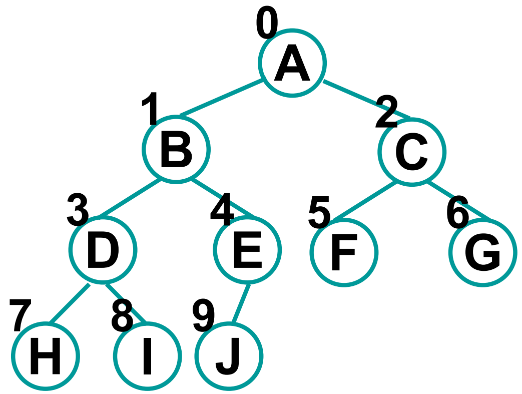
Array Representation of Binary Tree

A numbering scheme:



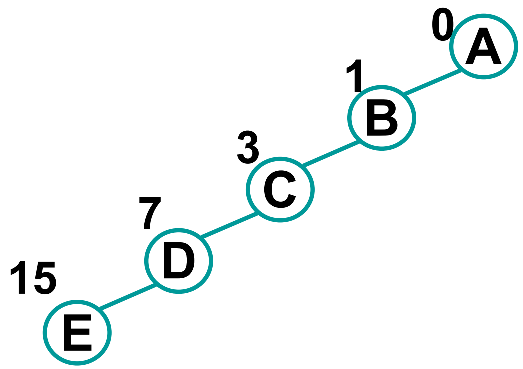
We can **represent binary trees using array** by applying this numbering scheme.

Example 1:



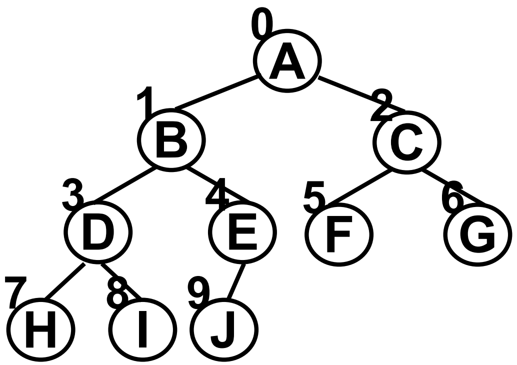
0	1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I	J

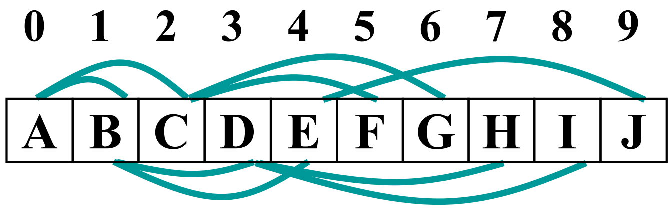
Example 2:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Array Representation of Binary Tree





Children of a node at slot ***i***:

Left(i) = 2i+1
Right(i) = 2i+2

Parent of a node at slot ***i***:

Parent(i) = $\lfloor$ (i-1)/2 $\rfloor$

$\lfloor x \rfloor$: **"Floor" The greatest integer less than x**

$\lceil x \rceil$: **"Ceiling" The least integer greater than x**

For any slot ***i***,

If ***i*** is odd: it represents a left son.

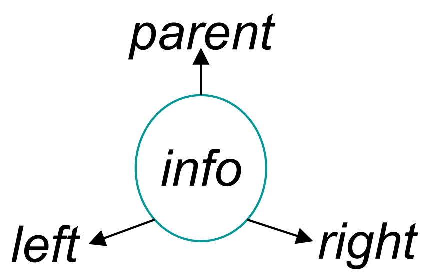
If ***i*** is even (but not zero): it represents a right son.

The node at the right of the represented node of ***i*** (if any), is at ***i***+1.

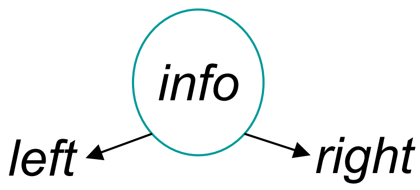
The node at the left of the represented node of ***i*** (if any), is at ***i***-1.

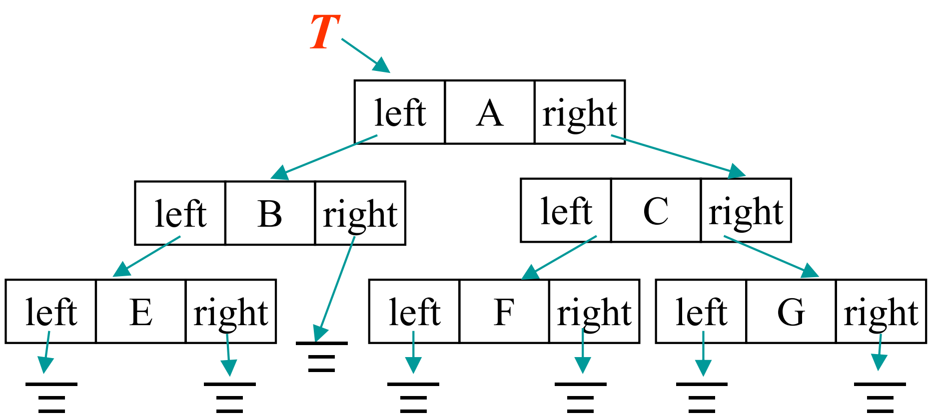
Linked Representation of Binary Tree

- Each node can contain *info, left, right, parent* fields
- where *left, right, parent* fields are node pointers pointing to the node's left son, right son, and parent, respectively.



- If the tree is always traversed in downward fashion (from root to leaves), the parent field is unnecessary.






```
class TreeNode
{
private:
        int info;
        TreeNode* left;
        TreeNode* right;
};
class Mytree
{
private:
        TreeNode* root;
}
```

- If the tree is empty, root = NULL; otherwise from root you can find all nodes.
- root->left and root->right point to the left and right subtrees of the root, respectively.

Link Representation of Binary Tree

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

class TreeNode
{
private:
      int info;
      TreeNode* left;
      TreeNode* right;
public:
      TreeNode();

      //create a new left child of a given node
      void SetLeft(int value) {..}

      //create a new right child of a given node
      void SetRight(int value) {..}

      void Insert(int );
}
```

```
class Mytree
{
private:
      TreeNode* root;
public:
      Tree();
      GetHeight();
      Compare(Mytree*);
      void InsertNode(int )
      void PreorderTraversal();
      void PreorderHelper(TreeNode*);
      void InorderTraversal();
      void InorderHelper(TreeNode*);
      void PostorderTraversal();
      void PostorderHelper(TreeNode*);
}
```

Array Representation of Binary Tree

- **Application: Find all duplicates in a list of numbers.**

Method 1: Compare each number with those before (or after) it.

e.g., to find all duplicates in <*7 4 5 9 5 8 3 3*>, we need to compare:

4 with 7

5 with 7,4

9 with 7,4,5

5 with 7,4,5,9

8 with 7,4,5,9,5

3 with 7,4,5,9,5,8

3 with 7,4,5,9,5,8,3

Method 2: Use a special binary tree (Binary Search Tree), T:

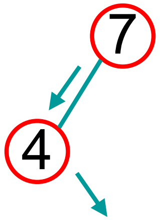
- Read number by number.
- Each time compare the number with the contents of T.
- If it is found duplicated, then output, otherwise add it to T.

Array Representation of Binary Tree

- Using Binary Search Tree to Find All Duplicates in a List of Numbers

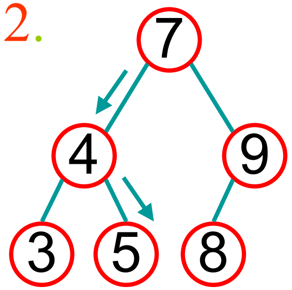
Indeed, searching and insertion are very quick in a binary search tree:

Example 1. The steps to insert a '5':



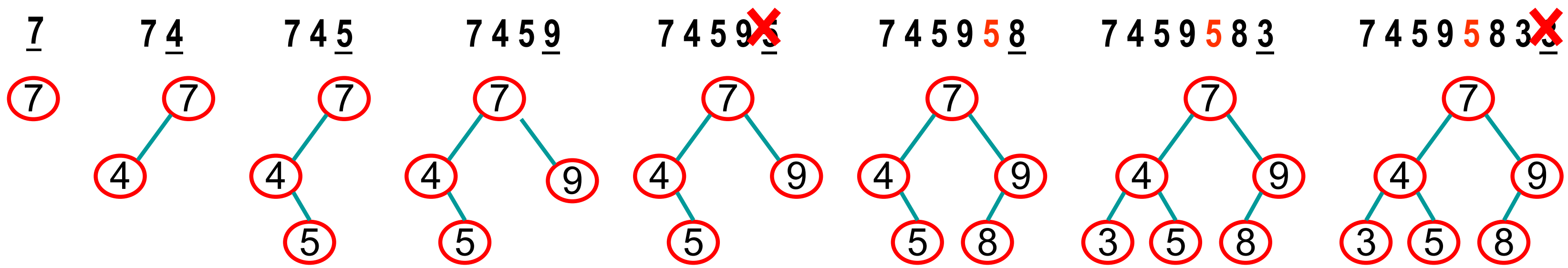
1. Compare '5' with the root. '5' is smaller than the root, so,
2. Go to left subtree, which has root = '4', '5' is larger than '4', so,
3. Go to the right subtree of '4', which is empty. => insert here.

Example 2.



The steps to insert a '5':

1. <same as example 1.>
2. <same as example 1.>
3. Go to the right subtree of '4', which has the root = '5'. Found=>no need to insert.



No. of comparisons required: 0 1 2 1 3 2 ___ ___

Array Representation of Binary Tree

➢ Using Binary Search Tree to Find All Duplicates in a List of Numbers

Method 2 - Use a special binary search tree (Binary Search Tree), T:

- Read number by number.
- Each time compare the number with the contents of **T**.
- If it is found duplicated, then output, otherwise add it to **T**.

Exercise:

Create a binary search tree according to the input sequence:

<*3, 5, 0, 2, 7, 9, 6, 8*>

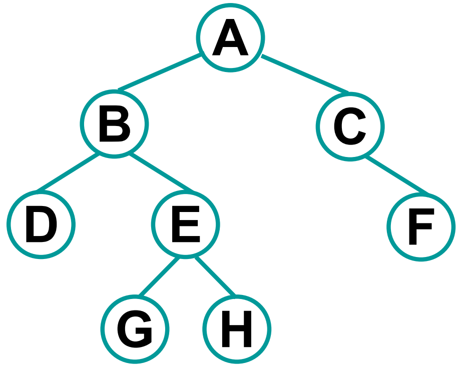
Binary Tree Operations - height

Review:

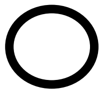
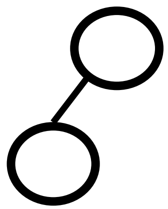
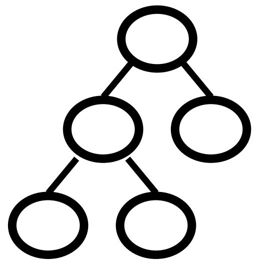
Depth of node : The depth of root(T) is zero.

The depth of any other node is one larger than his parent's depth.

Height of a tree: The maximum depth of any leaf in the tree.



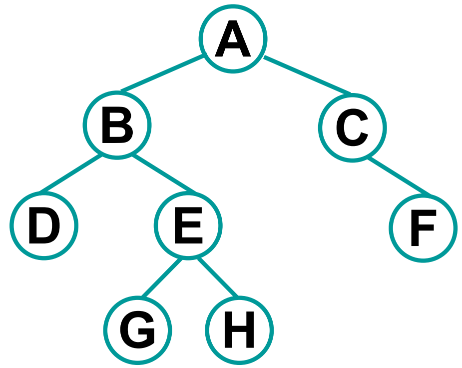
Example:

Height of a NULL binary tree is -1 or 0	 Height of a tree with 1 node is 0	 Height = 1	 Height = 2

Binary Tree Operations - height

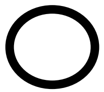
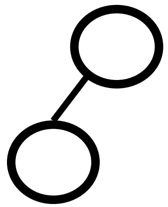
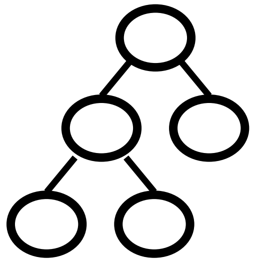
```
//To determine the height of a binary tree
int Mytree::height() {
        return root->height();
}
```

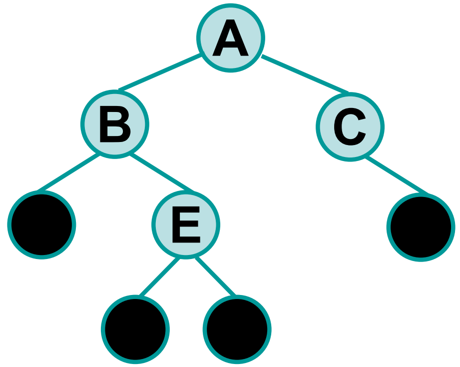
```
int TreeNode::height( )
{       int HeightOfLeftSubTree, HeightOfRightSubTree;
        if (this == NULL)
                return(0);
        if ((this->left == NULL) && (this->right == NULL))
                return(0); // the subroot is at level 0
        HeightOfLeftSubTree = this->left->height();
        HeightOfRightSubTree = this->right->height();
        if (HeightOfLeftSubTree > HeightOfRightSubTree)
                return___HeightOfLeftSubTree + 1__;
        else
                return___HeightOfRightSubTree + 1_;
}
```



Binary Tree Operations - countleaves

Example:

A NULL binary tree has 0 leaf node	 A tree with 1 node has 1 leaf node	 No. of leaf nodes = 1	 No. of leaf nodes = 3

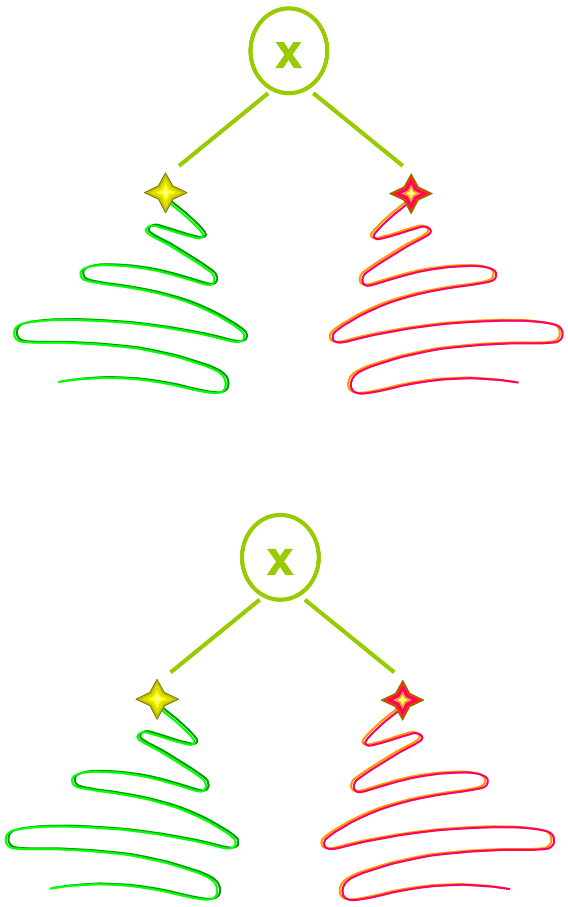



```
//To count the number of leaf nodes
int Mytree::count_leaf(TreeNode* p)
{
    if (p == NULL)
        return(0);
    else if ((p->left == NULL) && (p->right == NULL))
        return(1);
    else
        return(count_leaf(p->left) + count_leaf(p->right));
}
```

Binary Tree Operations - equal

```
// To compare 2 binary trees
bool Mytree::equal(Mytree* T)
{
        return root->equal(T->root);
}
```

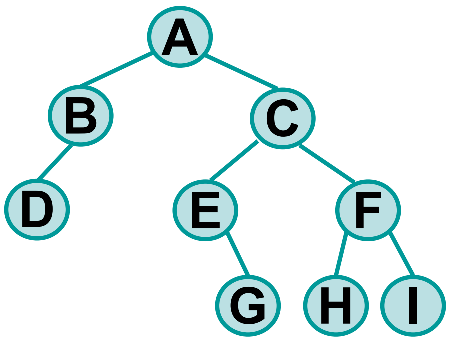
```
bool TreeNode::equal(TreeNode* TN)
{       if ((this == NULL) && (TN == NULL))
                return(true) ;

        if ((this != NULL) && (TN == NULL))
                return(false);
        if ((TN != NULL) && (this == NULL))
                return(false);

        if (this->info == TN->info)
                if (this->left->equal(TN->left) &&
                        this->right->equal(TN->right))
                            return(true);

        return(false);
}
```



Traversing Binary Tree

Traversing / walking through

A method of examining the nodes of the tree systematically so that each node is visited exactly once.



Three principle ways:

When the binary tree is empty, it is “traversed” by doing nothing, otherwise:

preorder traversal

1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree

A B D C E G F H I

inorder traversal

1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree

D B A E G C H F I

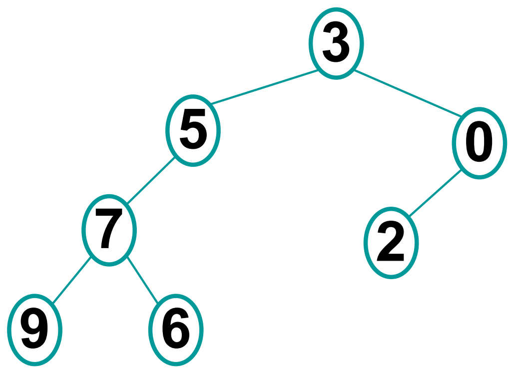
postorder traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root

D B G E H I F C A

Exercise 1

Showing the preorder, inorder and postorder traversals of the tree:



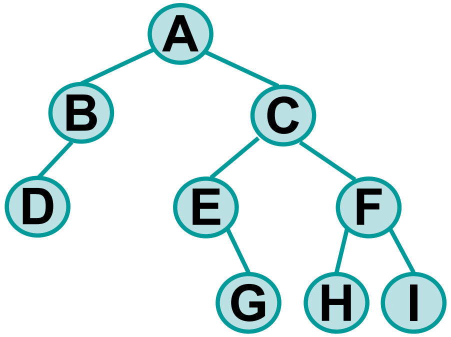
preorder:

inorder:

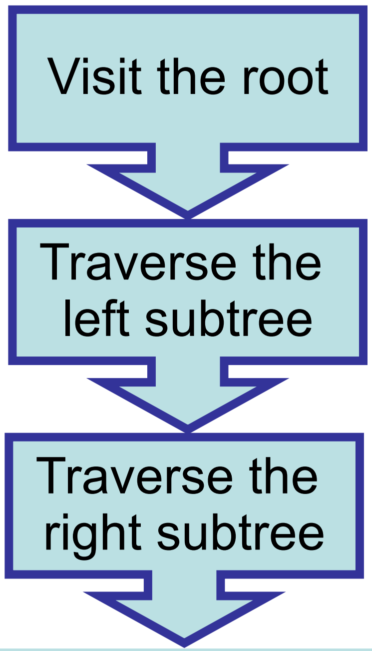
postorder:

Traversing Binary Tree

When the binary tree is empty, it is "traversed" by doing nothing, otherwise:



preorder traversal



A B D C E G F H I

Result:
= A (A's left) (A's right)
= A B **(B's left)** (B's right = NULL) (A's right)
= A B **(B's left)** (A's right)
= A B D (D's left=NULL) (D's right = NULL) (A's right)
= A B D (A's right)
= A B D C **(C's left)** (C's right)
= A B D C **E (E's left=NULL) (E's right)** (C's right)
= A B D C **E (E's right)** (C's right)
= A B D C **E G (G's left=NULL) (G's right = NULL)** (C's right)
= A B D C **E G** (C's right)
= A B D C **E G** F (F's left) (F's right)
= A B D C **E G** F H (H's left=NULL) (H's right =NULL) (F's right)
= A B D C **E G** F H I (I's left=NULL) (I's right =NULL)
= A B D C **E G** F H I

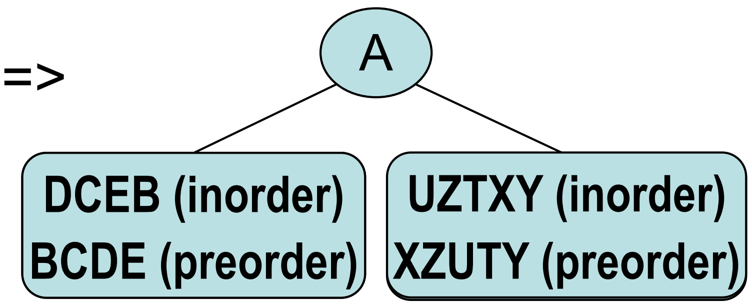
Traversing Binary Tree

Reconstruction of Binary Tree from its preorder and Inorder sequences	**Example:** Given the following sequences, find the corresponding binary tree: preorder : ABCDEXZUTY inorder : DCEBAUZTXY

Looking at the whole tree:

- "preorder : **A**BCDEXZUTY"
  ==> A is the root.
- Then, "inorder : DCEB<u>**A**</u>UZTXY"

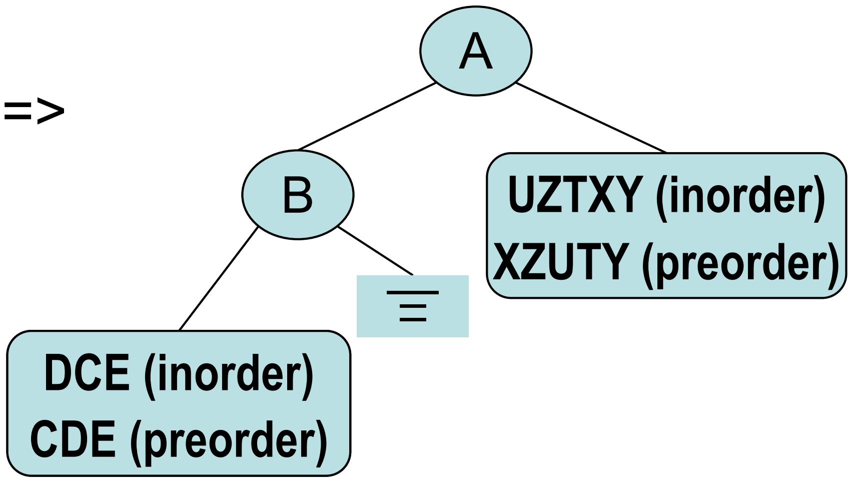
==>



Looking at the left subtree of A:

- "preorder : BCDE"
  ==> B is the root
- Then, "inorder: DCE<u>**B**</u>"

=>



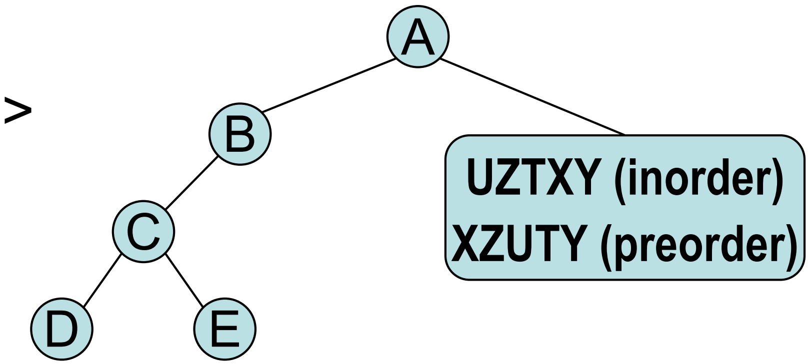
Traversing Binary Tree

Reconstruction of Binary Tree from its preorder and Inorder sequences	**Example:** Given the following sequences, find the corresponding binary tree: preorder : ABCDEXZUTY inorder : DCEBAUZTXY

Looking at the left subtree of B:

- "preorder : CDE"
  ==> C is the root
- Then, "inorder: D**<u>C</u>**E"

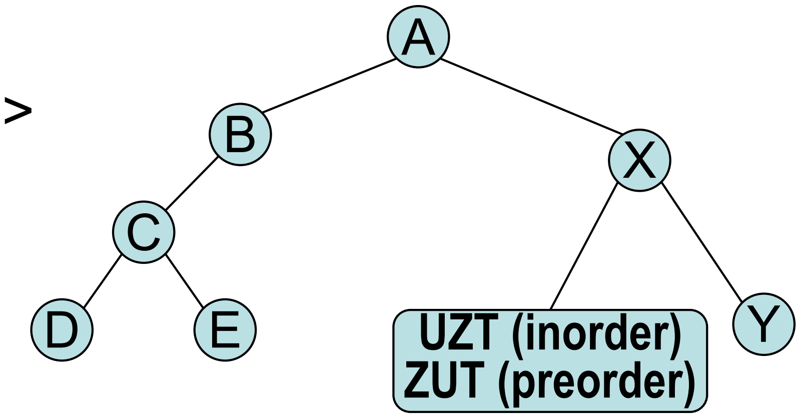
=>



Looking at the right subtree of A:

- "preorder : XZUTY"
  ==> X is the root
- Then, "inorder: UZT**<u>X</u>**Y"

=>



Traversing Binary Tree

Reconstruction of Binary Tree from its preorder and Inorder sequences

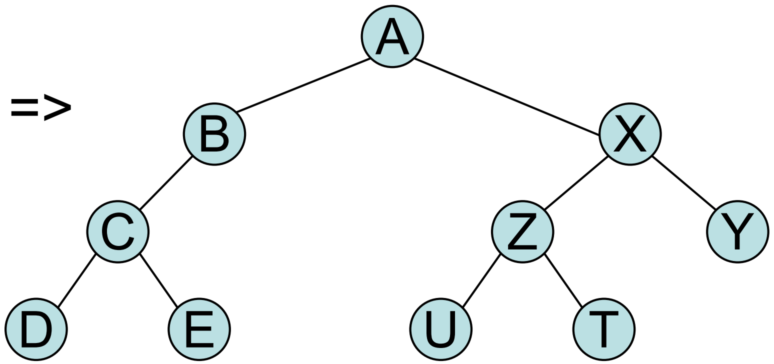
Example: Given the following sequences, find the corresponding binary tree:

preorder : ABCDEXZUTY
inorder : DCEBAUZTXY

Looking at the left subtree of X:

- "preorder : ZUT"
  ==> Z is the root
- Then, "inorder: U**Z**T"

=>



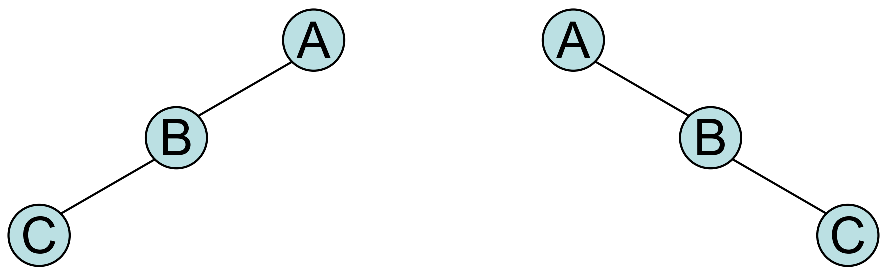
Traversing Binary Tree

But A binary tree may not be uniquely defined by its preorder and postorder sequences.

Example: **Preorder sequence: ABC**

Postorder sequence: CBA

We can construct 2 different binary trees:

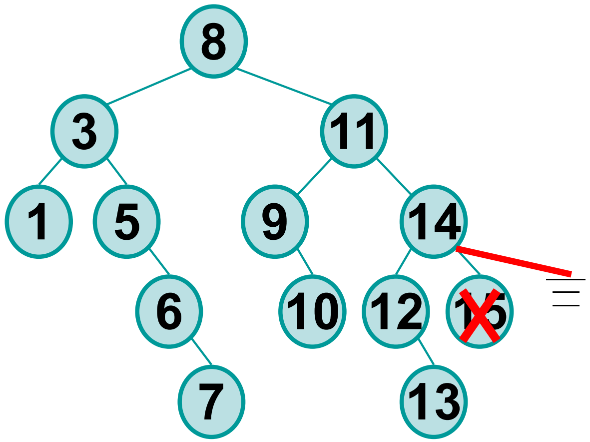


Binary Search Tree - Deletion

Delete a node in a BST: **void Mytree::delete(TreeNode* node)**

Case 1:

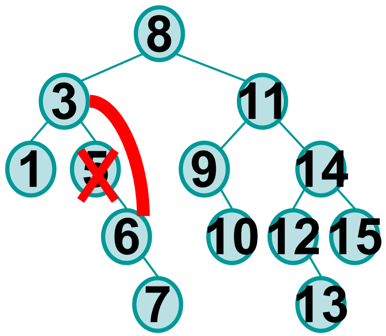
To delete a leaf node



- Set the parent node's child pointer to NULL

Case 2:

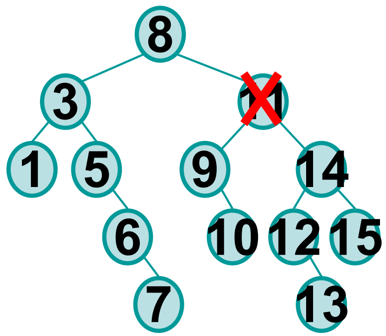
To delete a node that has 1 subtree (left or right)

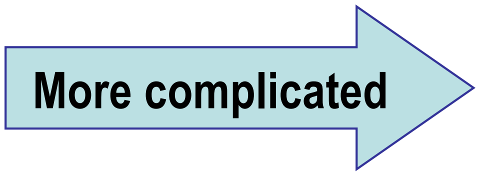


- Set the parent's child pointer to root of unwanted node's subtree (left or right).

Case 3:

To delete a node that has 2 subtrees





Binary Search Tree - Deletion

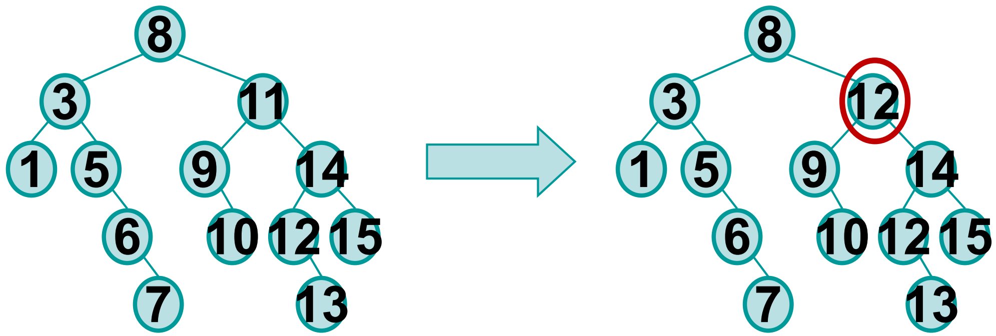
Delete a node in a BST: **void Mytree::delete(TreeNode* node)**

Case 3:

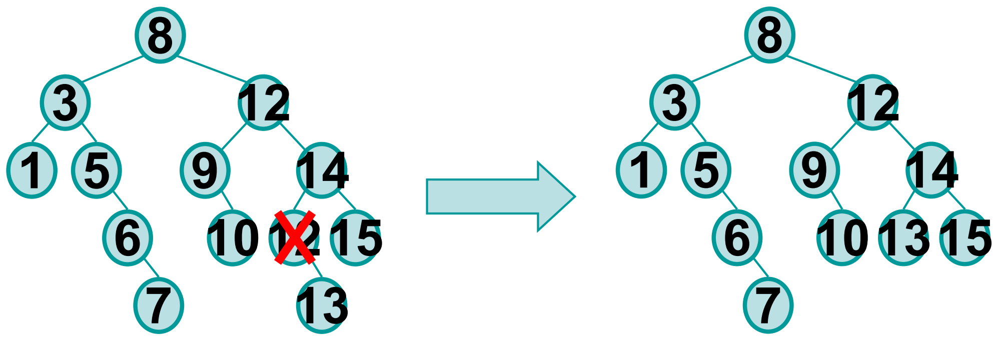
To delete a node that has 2 subtrees

- Replace its value by its inorder successor:

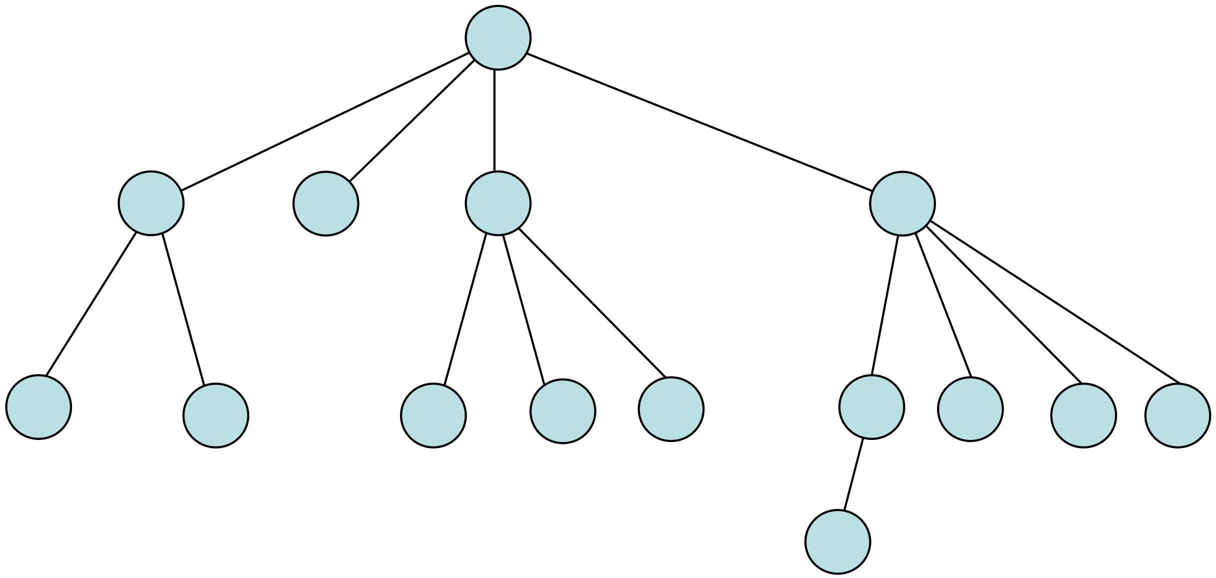
  The inorder successor is the leftmost node of right subtree.



- Delete the successor in turn:

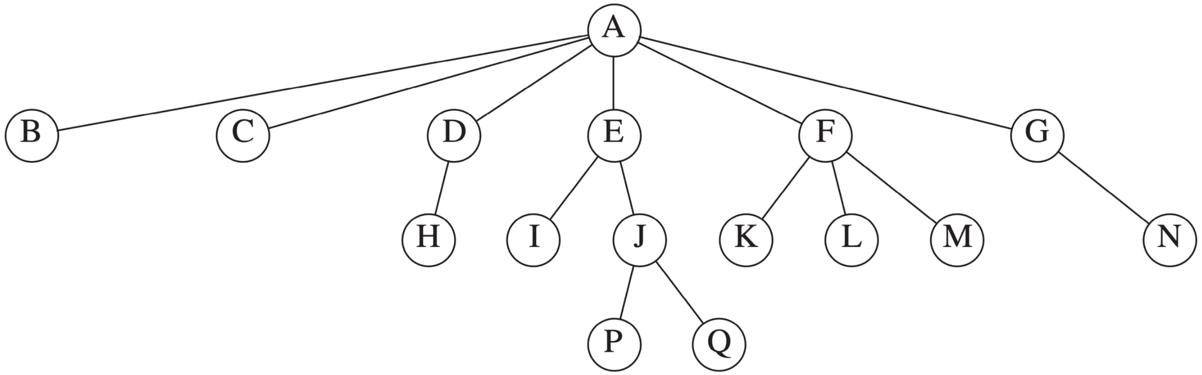


Binary Representation of General Tree



- One node can have many children nodes
- Impossible to make so many links
- Is there a way that each node uses only two links?
  - Link1:
  - Link2:

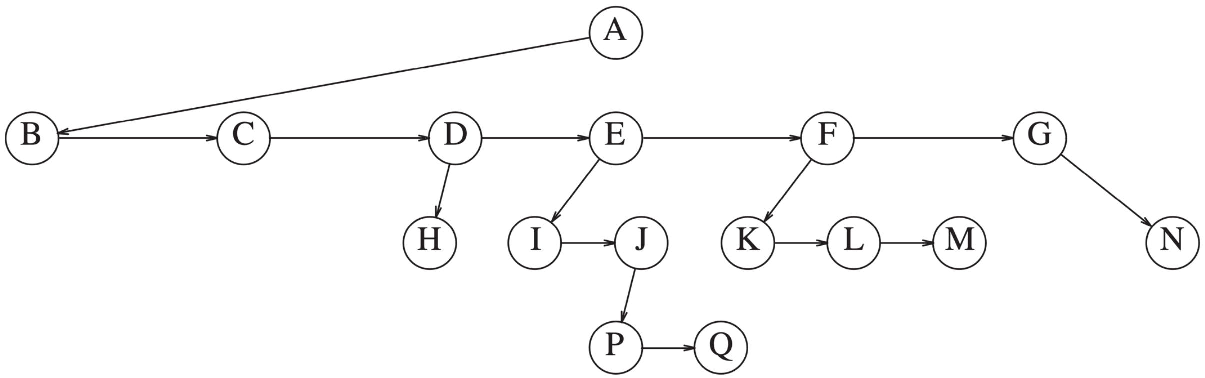
Binary Representation of General Tree



each node uses only two links

Link1: first child

Link2: next sibling




```
struct TreeNode
{
    Object    element;
    TreeNode *firstChild;
    TreeNode *nextSibling;
};
```

Learning Objectives

1. Explain the concept of Tree
2. Able to insert into and delete from a binary search tree
3. Able to do Tree Traversal; Able to reconstruct a tree given two suitable traversal orders
4. Able to write recursive functions on Tree

D:1;   C:1,2;   B:1,2,3;   A:1,2,3,4

Objective

- Heap
  - Heap-order
  - Heap operations
  - Applications

Heap

- Structure Property
  - A **complete** binary tree

- Heap-order Property (min-Heap)
  - The data in the root is smaller than the data in all the descendants of the root.

    Or
  - The data in a node is smaller than the data in its children

ADT of Heap

Value:

A sequence of items that belong to some data type ITEM_TYPE

Operations on heap h:

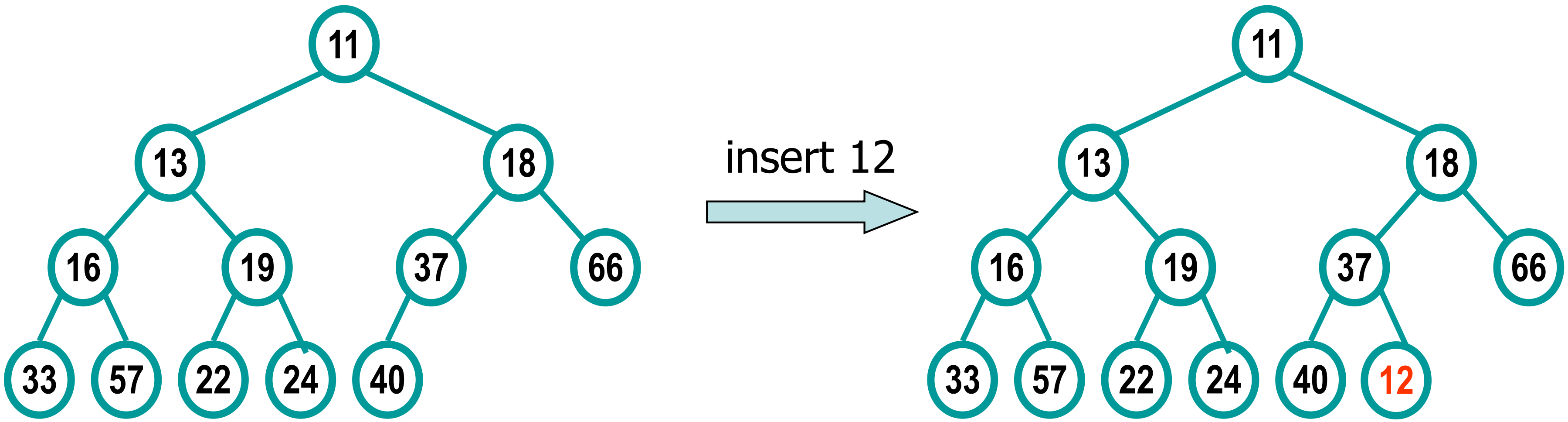
1. void ***Insert***(ITEM_TYPE e)

2. /*return the value stored in the root and then delete the root*/
   ITEM_TYPE ***DeleteMin***()

```
#define TOTAL_SLOTS 100
class MyHeap
{
    private:
        int size;
        int items[TOTAL_SLOTS];
    public:
        void Insert(int key);
        int DeleteMin();
};
```

Heap: Insert

To insert an item	Step 1: Put the item at the first empty slot in the array
	Step 2: Adjust the heap to satisfy heap-order property (float up)

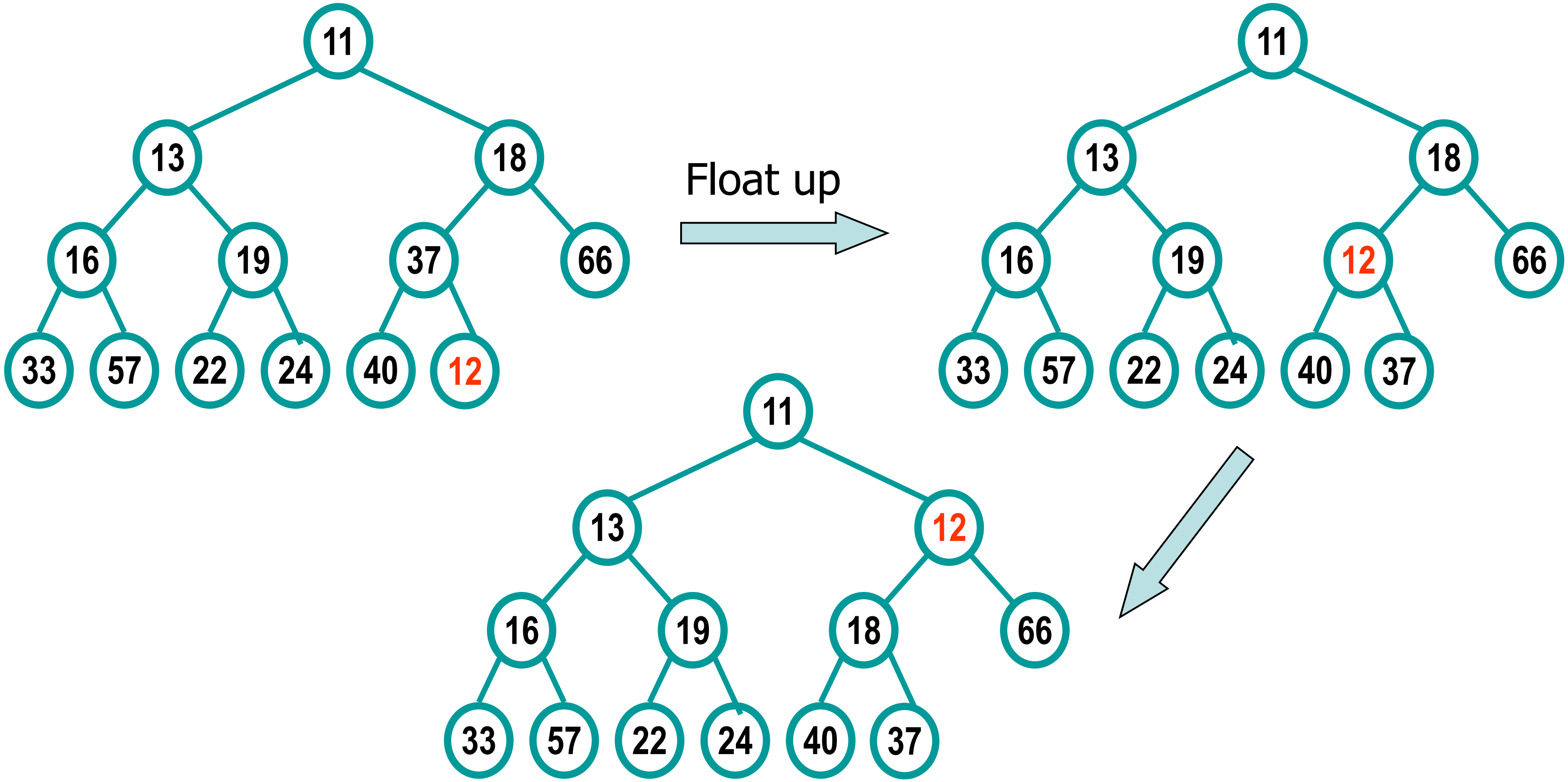


Heap: Insert

To insert an item

Step 1: Put the item at the first empty slot in the array

Step 2: Adjust the heap to satisfy heap-order property (float up)



Heap: Insert

```
void MyHeap::Insert(int key)
{
	int temp;
	int i=size;
	items[i]=key; // the first empty slot
	while (items[i]<items[(i-1)/2] && i!=0)
	{
		temp=items[i];
		items[i]=items[(i-1)/2];
		items[(i-1)/2]=temp;
		i=(i-1)/2;
	}
	size++;
}
```

Children of a node at slot ***i***:

Left(i) = 2i+1
Right(i) = 2i+2

Parent of a node at slot ***i***:

Parent(i) = $\lfloor (i-1)/2 \rfloor$

$\lfloor x \rfloor$: "Floor" The greatest integer less than x

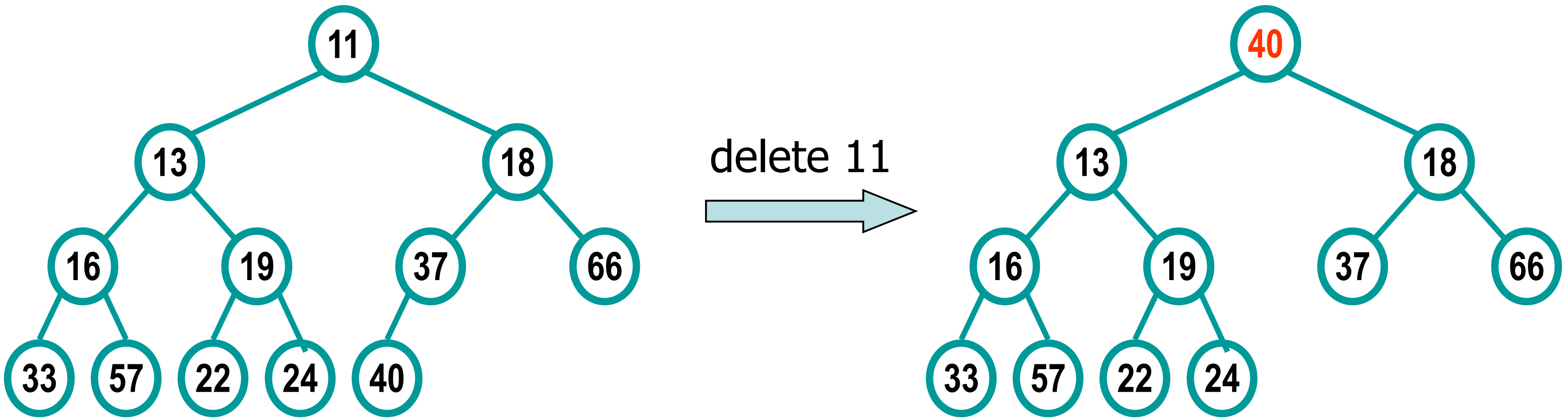
$\lceil x \rceil$: "Ceiling" The least integer greater than x

Heap: DeleteMin

To delete the Min

Step 1: Copy the last data to the root

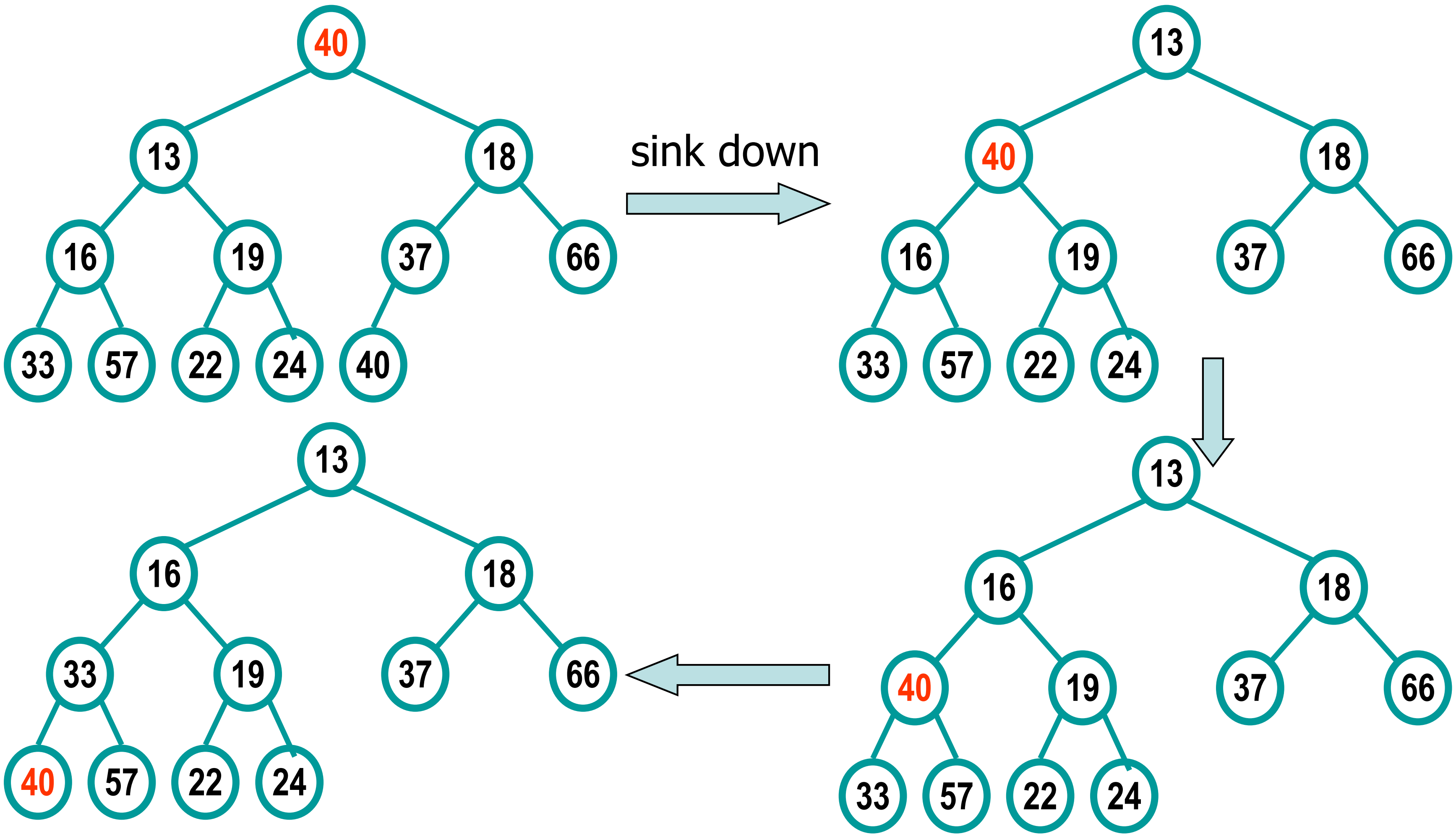
Step 2: Adjust the heap to satisfy heap-order property (sink)



Heap: DeleteMin

To delete the Min Step 1: Copy the last data to the root

Step 2: Adjust the heap to satisfy heap-order property (sink)



Heap: DeleteMin

Any bugs?

```
int MyHeap::DeleteMin()
{
    int temp, value;
    int i=size;
    value=items[0];
    items[0]=items[i-1];
    int hole=0;
    int temp=items[hole];
    for ( ; hole*2+1<=size; hole=child)
    {
        child=hole*2+1; // left
        if(items[child+1]<items[child])
            child++;
        if(items[child]<temp)
            items[hole]=items[child];
        else
            break;
    }

    items[hole]=temp;
    size--;
    return value;
}
```

Other Operations

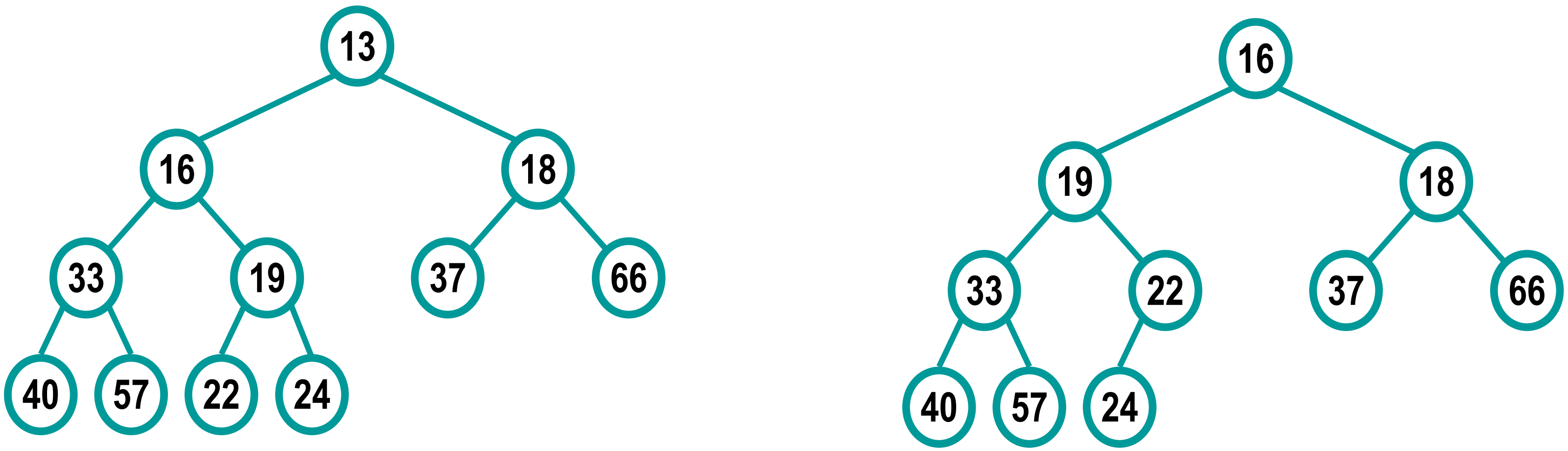
- IncreaseKey(int p, int amount)
- DecreaseKey(int p, int amount)

Exercises:

1. Given an array of data, how to build a min-heap?
   3,9,7,6,1,4,2,5 (showing each step)

2. What’s the possible applications for heap?

Exercise 3



How does it look like after providing DeleteMin()?

Review about Linked Lists

- Abstract Data Types
- Pointers and References
- Singly Linked List
- Circular Lists
- Doubly Linked Lists
- Applications

Review: Abstract Data Type

The ***set*** ADT consists of 2 parts:

1. Definition of values

involves

- definition
- condition (optional)

2. Definition of operations

each operation involves

- header
- precondition (optional)
- postcondition

Value:

A set of elements

Condition: elements are distinct.

Operations for Set *s:

1. void Add(ELEMENT e)

postcondition: e will be added to **s*

2. void Remove(ELEMENT e)

precondition: *e* exists in **s*

postcondition: *e* will be removed from **s*

3. int Size( )

postcondition: the no. of elements in **s* will be returned.

…

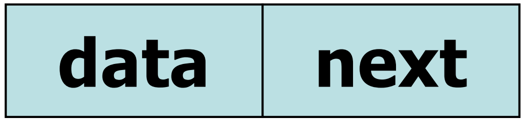
Review: List

A Linear List (or **a list**, for short)

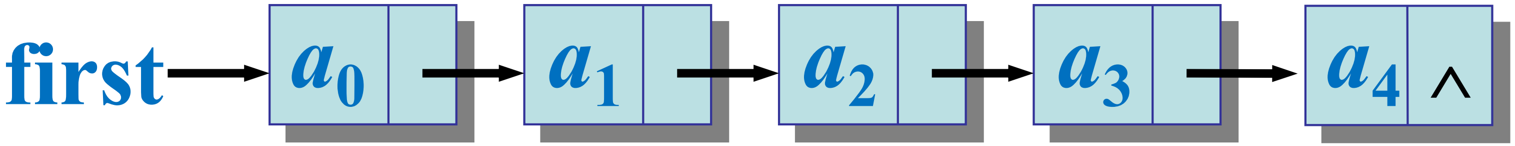
- is a sequence of *n* nodes $\{x_1, x_2, .., x_n\}$ whose essential structural properties involve only the relative positions between items as they appear in a line.
- can be implemented as
  - Arrays: statically allocated or dynamically allocated
  - Linked Lists: dynamically allocated
- A list can be sorted or unsorted.

Review: Singly Linked List

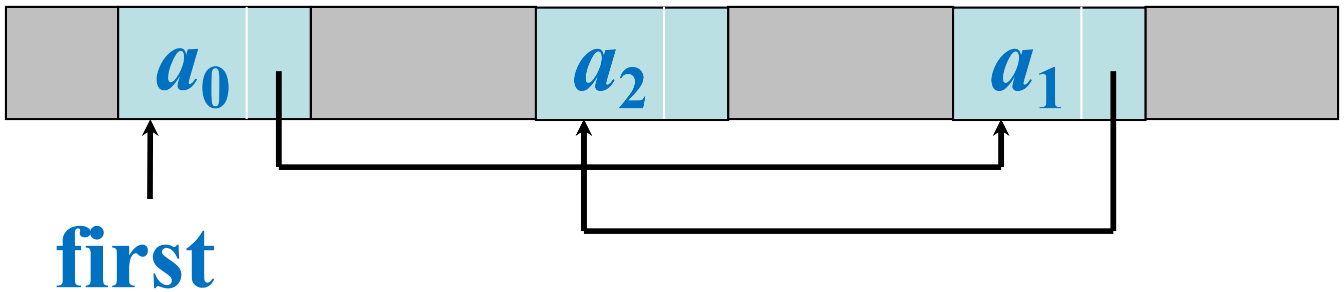
- Each item in the list is a node



- Linear Structure



- Node can be stored in memory consecutively /or not (logical order and physical order may be different)



Review: Singly Linked List

```
// List.h
#include <string>
using namespace std;

class ListNode
{
public:
        ListNode( int );
        ListNode( int, ListNode *);
        ListNode *get_Next()
        {
                return next;
        }
        …
private:
        int data;
        ListNode *next;
};
```

```
class List
{
public:
    List( String  );
    List();
//various member functions
private:
    ListNode *first;
    string name;
}
```

Review: Singly Linked List

- Operations:
  - InsertNode: insert a new node into a list
  - RemoveNode: remove a node from a list
  - SearchNode: search a node in a list
  - CountNodes: compute the length of a list
  - PrintContent: print the content of a list
  - …
- All the variables are defined to be “int”, how about when we want to use “double”?
  - Write a different class of list for “double”? Or…

Review: Search for a node

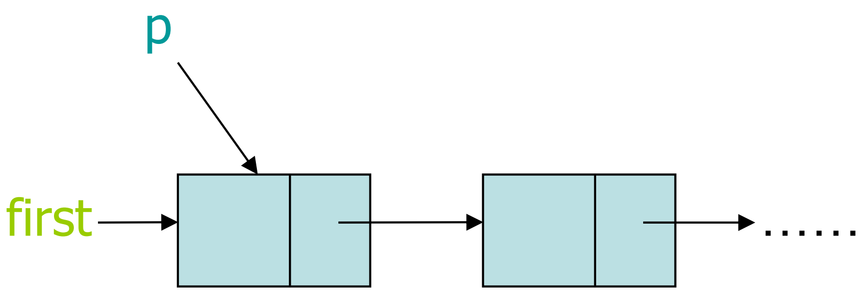
Use a pointer p to traverse the list

- If found: return the pointer to the node,
- otherwise: return NULL.

```cpp
//List.cpp
ListNode* List::Search(int data)
{
        ListNode *p=first;
        while (p!=NULL)
        {
                if (p->getData()==data)
                        return p;
                p=p->getNext();
        }
        return NULL;
}
```

```cpp
// List.h
#include <string>
using namespace std;

class ListNode
{
public:
      ListNode( int );
      ListNode( int, ListNode *);
      ListNode *get_Next()
      {
            return next;
      }
      ...
private:
      int data;
      ListNode *next;
};
```

```cpp
class List
{
public:
      List( String  );
      List();
//various member functions
private:
      ListNode *first;
      string name;
}
```



Review: Advantages / Disadvantages of Linked List

Linked allocation: Stores data as individual units and link them by pointers.

Advantages of linked allocation:

- Efficient use of memory
  - Facilitates data sharing
  - No need to pre-allocate a maximum size of required memory
  - No vacant space left
- Easy manipulation
  - To delete or insert an item
  - To join 2 lists together
  - To break one list into two lists
- Variations
  - Variable number of variable-size lists
  - Multi-dimensional lists
    - (array of linked lists, linked list of linked lists, etc.)
- Simple sequential operations (e.g. searching, updating) are fast

Disadvantages:

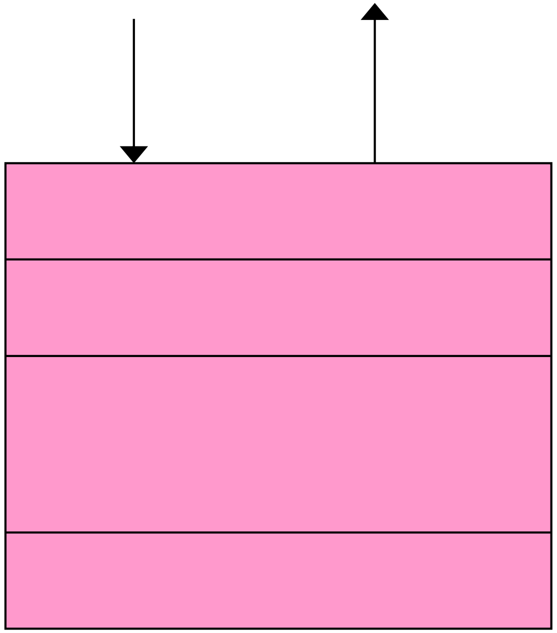
- Take up additional memory space for the links
- Accessing random parts of the list is slow. (need to walk sequentially)

Review: Objective of Lec 2

- Stack Abstract Data Type
- Sequential Allocation
- Linked Allocation
- Applications

Review: Stack

- Stack is a list with the restriction that insertions and deletions (usually all the accesses) can only be performed at one end of the list
- Also known as: Last-in-first-out (LIFO) list



Review: ADT of Stack

Value:

A sequence of items that belong to some data type ITEM_TYPE

Operations for a stack *s*:

1. Boolean IsEmpty()

   Postcondition: If the stack is empty, return true, otherwise return false

2. Boolean IsFull()

   Postcondition: If the stack is full, return true, otherwise return false

3. ITEM_TYPE Pop() /*take away the top one and return its value*/

   Precondition: s is not empty

   Postcondition: The top item in s is removed from the sequence and returned

4. ITEM_TYPE top() /*return the top item's value*/

   Precondition: s is not empty

   Postcondition: The value of the top item in s is returned

5. Void Push(ITEM_TYPE e) /*add one item on top of the stack*/

   Precondition: s is not full

   Postcondition: e is added to the sequence as the top one

Review: Array Implementation of Stack

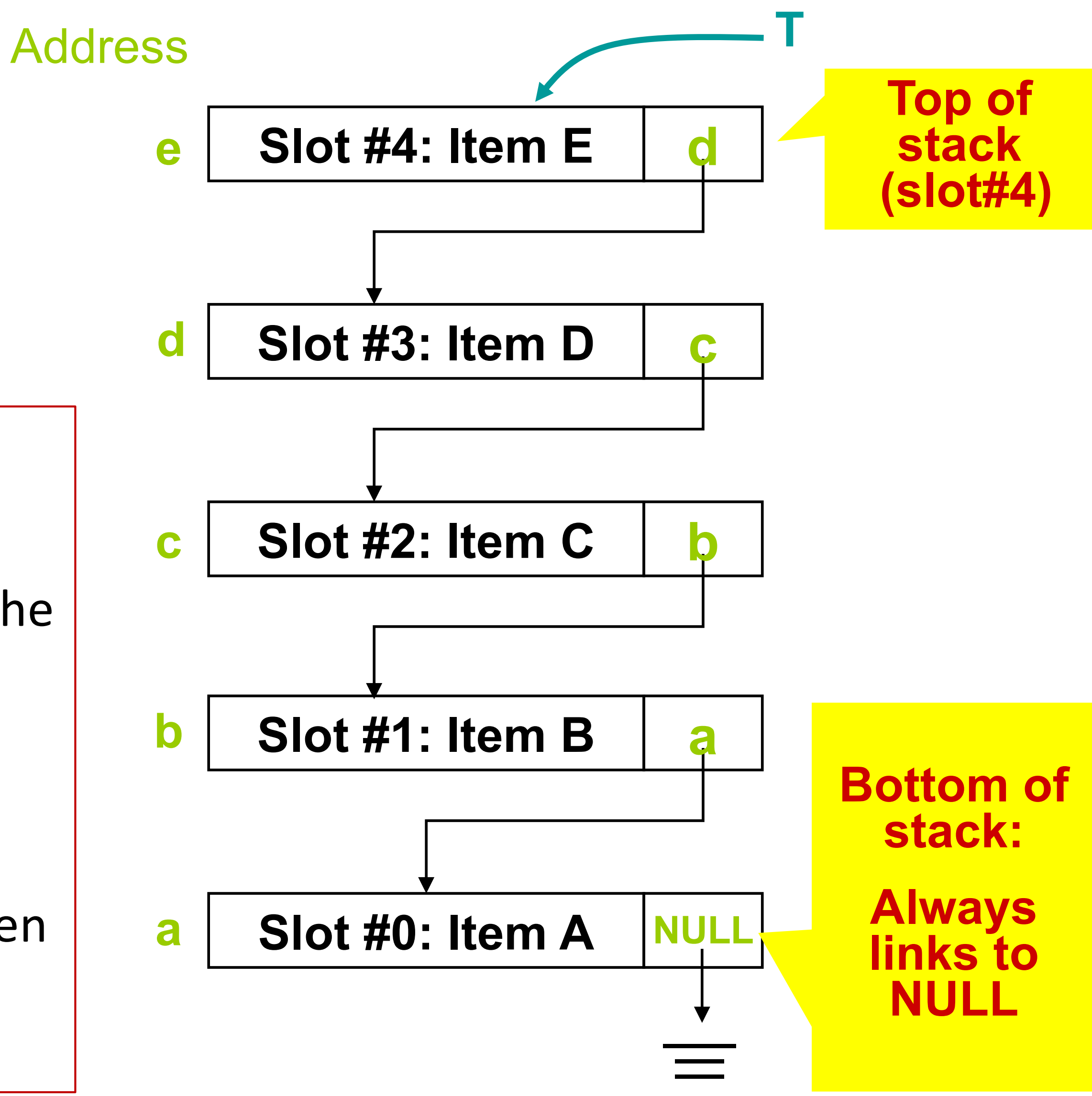
```
// MyStack.h
#include “stdlib.h”
{
	public class MyStack
	{
		public:
				MyStack( int );
				bool IsEmpty();
				bool IsFull();
				void push(int );
				int pop();
				int top();
		private:
				int* data;
				int top;
				int MAXSize;
	};
}
```

```
// MyStack.cpp

#include “MyStack.h”
MyStack::MyStack(int size)
{
		data=new int[size];
		top=-1;
		MAXSize=size;
}
bool MyStack::IsEmpty()
{
		return (top==-1);
}
bool MyStack::IsFull()
{
		return (top==MAXSize-1);
}
```

Review: Linked Implementation of Stack

Stack can also be implemented with **linked list**.

- Typically, a pointer points to the top of the stack. (T)
- When the stack is empty, this pointer will be NULL.
- Each slot is allocated only when it is needed to store an item.



Review: Linked Implementation of Stack

```
// MyStack.h

#include "stdlib.h"
#include "ListNode.h"
{
        class MyStack
        {
        public:
                MyStack( );
                Pop();
                IsEmpty();
                Push(int );
                …
        private:
                ListNode *Top;
        };
}
```

```
// ListNode.h

#include "stdlib.h"
{
        class ListNode
        {
        public:
                ListNode( int );
                ListNode( int, ListNode *);
                ListNode *get_Next()
                {
                        return next;
                }
                …
        private:
                int data;
                ListNode *next;
        };
}
```

Review: Application 2: Balancing Symbols

- When writing programs, we use
  - () parentheses [] brackets {} braces
- A lack of one symbol may cause the compiler to emit a hundred lines without identifying the real error
- Using stack to check the balance of symbols
  - [( )] is correct while [( ]) is incorrect

- Read the code until end of file
  - **If** the character is an opening symbol: ([{, **then** push it onto the stack
  - **If** the character is a closing symbol:)] }, **then** pop one (if the stack is not empty) from the stack to see whether it is the correct correspondence
  - Output error in other cases

Review: Application 3 Evaluation of Postfix Expression

- Infix Expression Example: (A+B)*((C-D)*E+F)

  We need to add "(" and ")" in many cases.

- Postfix Expression Example: AB+CD-E*F+*

  Each operator follows the two operands.

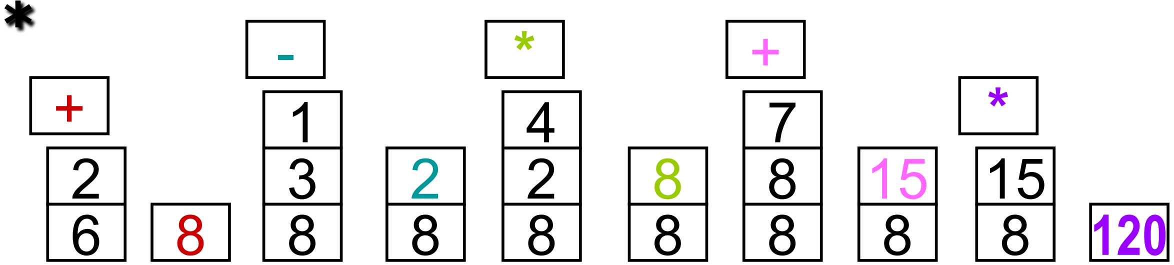
  The order of the operators (left to right) determines the actual order of operations in evaluating the expression.

- Prefix expression Example : *+AB+*-CDEF

  Each operator precedes the two operands.

Review: Application 3 Evaluation of Postfix Expression

6 2 + 3 1 - 4 * 7 + *
=8 3 1 - 4 * 7 + *
=8 2 4 * 7 + *
=8 8 7 + *
=8 15 *
=120



The method:

- Scan the expression from left to right.
- For each symbol, if it is an operand, we store them for later operation (LIFO) push
- If the symbol is an operator, take out the latest 2 operands stored and compute with the operator. pop pop

  Treat the operation result as a new operand and store it. push
- Finally, we can obtain the result as the only one operand stored. pop

Review: Program Complexities

- Algorithms
- Asymptotic Notation
- Asymptotic Performance
- Analyze program complexities

Review: Asymptotic Notation

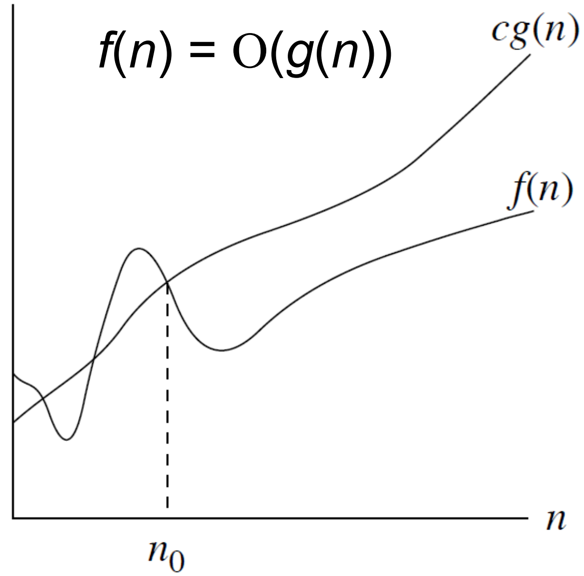
- How can we indicate running times of algorithms?
- Need a notation to express the growth rate of a function
- A way to compare "size" of functions:
    - O-notation ("Big-oh") $\approx \leq$ (upper bound)
    - Ω-notation ("Big-omega") $\approx \geq$ (lower bound)
    - Θ-notation ("theta") $\approx =$ (sandwich)

Review: O -notation (1/2)

- O-notation provides an **asymptotic upper bound** of a function.
- For a given function $g(n)$, we denote $O(g(n))$

(pronounced “big-oh” of g of n) by the set of functions:

$O(g(n))$ = { $f(n)$: there exist **positive** constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }



Review: O -notation (2/2)

- We write $f(n) = \mathrm{O}(g(n))$ to
  - Indicate that $f(n)$ is a member of the set $\mathrm{O}(g(n))$
  - Give that $g(n)$ is an upper bound for $f(n)$ to within a constant factor

- Example: $2n^2 = \mathrm{O}(n^3)$, with $c = 1$ and $n_0 = 2$
  - When $n = 1$: $2(1)^2 = 2 \leq (1)^3 = 1$ ☒
  - When $n = 2$: $2(2)^2 = 8 \leq (2)^3 = 8$ ☑
  - When $n = 3$: $2(3)^2 = 18 \leq (3)^3 = 27$ ☑

Review: Asymptotic Notation

- Relationship between typical functions
  - $\log n = o(n)$
  - $n = o(n \log n)$
  - $n^c = o(2^n)$ where n^c may be n^2, n^4, etc.
  - If $f(n)=n+\log n$, we call *log n* lower order terms

$\log n < \mathrm{sqrt}(n) < n < n\log n < n^2 < n^4 < 2^n < n!$

Review: Asymptotic Notation

- When calculating asymptotic running time
  - Drop low-order terms
  - Ignore leading constants

- Example 1: $T(n) = An^2+Bn+C$
  - An^2
  - $T(n) = O(n^2)$
- Example 2: $T(n) = An\log n+Bn^2+Cn+D$
  - Bn^2
  - $T(n) = O(n^2)$

Review: Asymptotic Performance

General rules for Big-Oh Analysis:

Rule 1. FOR LOOPS

The running time of a *for* loop is at most the running time of the statements inside the *for* loop (including tests) times no. of iterations

```
for (i=0;i<N;i++)
      a++;
```

O(N)

Rule 2. NESTED FOR LOOPS

The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

```
for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            k++;
```

$O(N^2)$

Rule 3. CONSECUTIVE STATEMENTS

Count the maximum one.

```
for (i=0;i<N;i++)
      a++;

for (i=0;i<N;i++)
      for (j=0;j<N;j++)
            k++;
```

$O(N^2)$

Rule 4. IF / ELSE

For the fragment:

```
If (condition)
      S1
else
      S2,
```

take the test + the maximum for S1 and S2.

Review: Recursive Relation

- $T(n)=T(n-1)+A$; $T(1)=1$
  - → $T(n)=O(n)$
- $T(n)=T(n-1)+n$; $T(1)=1$
  - → $T(n)=O(n^2)$
- $T(n)=2T(n/2) + n$; $T(1)=1$
  - → $T(n)=O(n \log n)$, why???
- More general form: $T(n)=aT(n/b)+cn$
  - Master's Theorem (You are not required to know)

$T(n) = 2T(n/2) + n$
$= 2(2T(n/4) + n/2) + n = 4T(n/4) + 2n + n$
$= 4(2T(n/8) + n/4) + 2n + n = 8T(n/8) + 4n + 2n + n$
$= \ldots$
$=$??? $T(1) +$??? $+ \ldots + 4n + 2n + n$

Review about Queues and Hashing

Queue

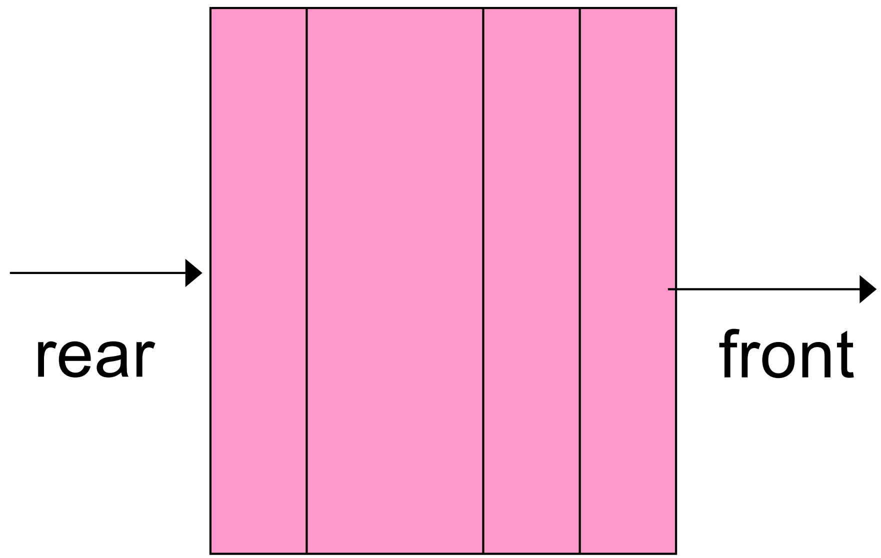
– Queue Abstract Data Type
– Sequential Allocation
– Linked Allocation
– Applications
– Priority Queues

Hashing

– Sparse Data
– Key Based Data
– Hash Table
– Hash Functions
– Collision Resolution
– Applications

Review: Queue

- Queue is a list with the restriction that insertions are performed at one end and deletions are performed at the other end of the list
- Also known as: First-in-first-out (FIFO) list



Review: ADT of Queue

Value:

A sequence of items that belong to some data type ITEM_TYPE

Operations on q:

1. Boolean IsEmpty()

   Postcondition: If the queue is empty, return true, otherwise return false

2. Boolean IsFull()

   Postcondition: If the queue is full, return true, otherwise return false

3. ITEM_TYPE Dequeue() /*take out the front one and return its value*/

   Precondition: q is not empty

   Postcondition: The front item in q is removed from the sequence and returned

4. Void Enqueue(ITEM_TYPE e) /*to append one item to the rear of the queue*/

   Precondition: q is not full

   Postcondition: e is added to the sequence as the rear one

Review: Implementation of Queue Sequential Allocation (Using Array)

```
#define TOTAL_SLOTS 100
class MyQueue
{
	private:
		int front; //the index of the front slot that contains the front item
		int rear;  //the index of the first empty slot at the rear of queue
		int items[TOTAL_SLOTS];
};
```

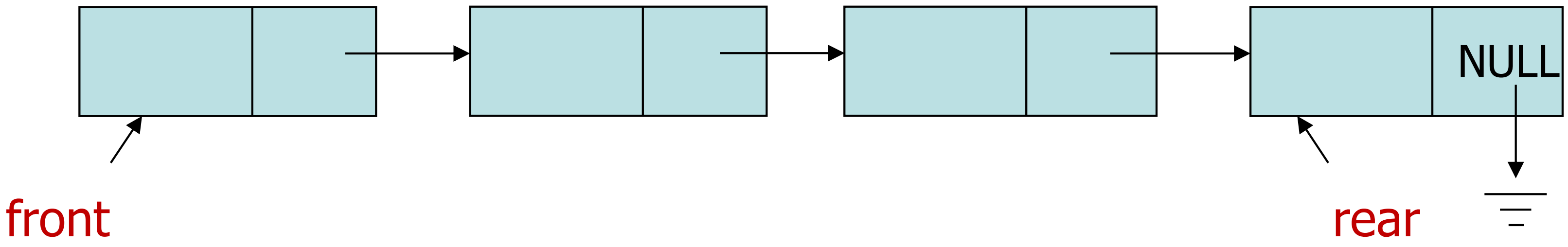
Slot#0	Slot#1	Slot#2	…	Slot#98	Slot#99

Suppose some items are appended into the queue:

Slot#0 Item A	Slot#1 Item B	Slot#2 Item C	Slot#3 Empty	… Empty	Slot#99 Empty

front **rear**

Review: Implementation of Queue Using Linked List



- Queue can also be implemented with linked list.
- A pointer front points to the first node of the queue.
- A pointer rear points to the last node of the queue.
- If the queue is empty, then front=rear=NULL.
- When will the queue be full?

Review: Priority Queue

Priority Queue

- The elements in a stack or a FIFO queue are ordered based on the sequence in which they have been inserted.
- In a priority queue, the sequence in which elements are removed is based on the priority of the elements.

Ordered Priority Queue

A Priority=1	B Priority=2	C Priority=3	D Priority=3

(highest priority) (lowest priority)

The first element to be removed.

Unordered Priority Queue

B Priority=2	C Priority=3	A Priority=1	D Priority=3

Review: Priority Queue

Priority Queue - List Implementation

- To implement a priority queue as an **ordered** list.

  Time complexity of the operations :

  (assume the sorting order is from highest priority to lowest)

  Insertion: Find the location of insertion. *O(n)*

  Link the element at the found location. *O(1)*

  Altogether: *O(n)*

  Deletion: The highest priority element is at the front.

  i.e., Remove the front element takes ***O(1)*** time

Review: Priority Queue

Priority Queue - List Implementation

- To implement a priority queue as an **unordered** list.

  Time complexity of the operations :

  Insertion: Simply insert the item at the rear. *O(1)*

  Deletion: Traverse the entire list to find the maximum priority element. *O(n)*.

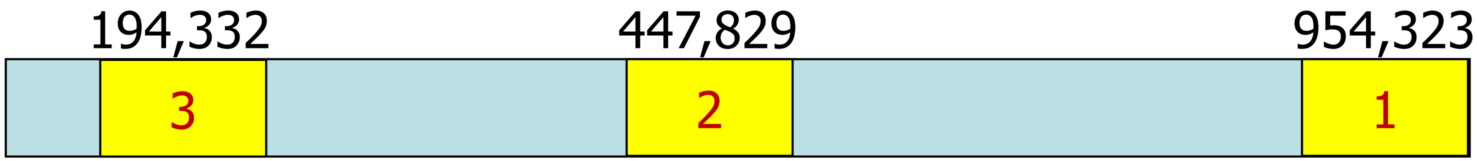
  Copy the value of the element to return it later. *O(1)*

  Delete the node. *O(1)*

  Altogether: *O(n)*

Review: Sparse data

- How to store those data in the computer so that we can easily get the player's information by their keys?
  - Array:
  - A lot of memory space wasted



  - Linked List:

    Hard to search if we have 10,000 players

  - Hash Table

    Best solution in this case!

Review: Hash Functions

Good hash function:

Fast computation, Minimize collision

Kinds of hash functions:

- **Division**: Slot_id = Key % table_size.
- **Others**: eg., Slot_id = (Key^2 + Key + 41) % table_size
- **table_size should better be a prime number.**

Review: Combination of Hash Functions

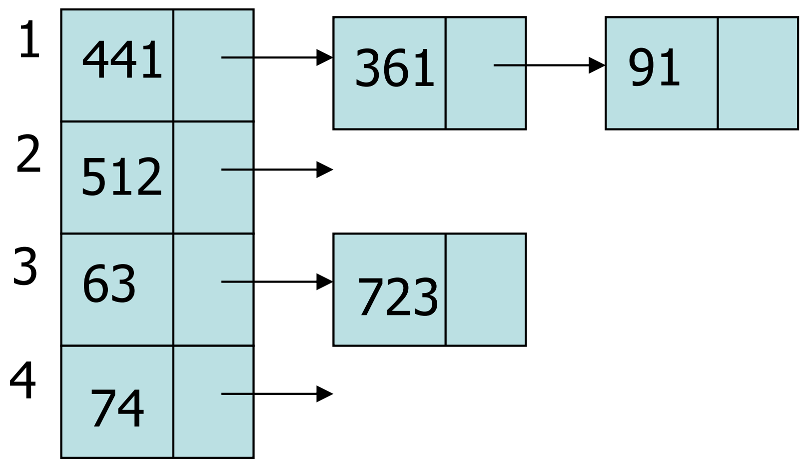
- Collision is easy to happen if we use ***%*** function
- Combination:
  - Apply hash function h_1 on key to obtain *mid_key*
  - Apply hash function h_2 on *mid_key* to obtain *Slot_id*
- Example:
  - We apply %101 on 12320324111220 and get 79
  - We apply %10 on the result 79 obtained by %101
    - 79 % 10 =9

Review: Collision Resolution - Open Addressing

- Linear Probing
  - If collide, try Slot_id+1, Slot_id+2
- Quadratic Probing
  - If collide, try Slot_id+1, Slot_id+2^2,…
- Double Hashing
  - If collide, try Slot_id+$h_2(x)$, Slot_id+$2h_2(x)$,… (prime size important)
- General rule: If collide, try other slots in a certain order
- How to find data?
  - If not found, try the next position according to different probing rule
  - Every key has a preference over all the positions
  - When finding them, just search in the order of their preferences

Review: Collision Resolution - Separate Chaining

- Problems with Open Addressing?
- Using linked list to solve Collision
  - Every slot in the hash table is a linked list
  - Collision→Insert into the corresponding list
  - Find data→Search the corresponding list

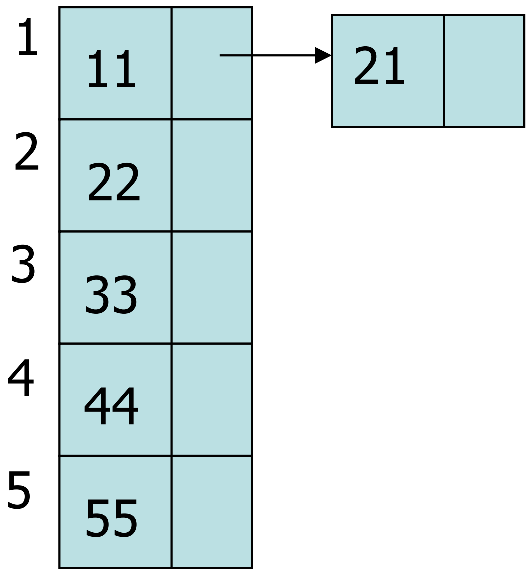


Review: Collision Resolution

- Example: 11,22,33,44,55,66,77,88,99,21
  - Using linear probing

21	11	22	33	44	55	66	77	88	99

  - Using separate chaining



Objective

- Definition and Terminology
- Binary Tree
  - Operations
  - Recursive functions
  - Traversal
- Binary Search Tree
  - Insertion, deletion
- Binary Representation of General Tree

Objective

- Heap
  - Heap-order
  - Heap operations
  - Applications

Check List

- Implementation of operations in Singly Linked List (count/insert/delete/search/…)?
- The output of a stack when it performs *push* and *pop* operations?
- Evaluation of Postfix Expression with stack?
- Big-O analysis for a given program?
- Big-O notations for an expression?
- Big-O complexity for recursive relations
- The output of a queue when it performs *enqueue* and *dequeue* operations?
- Insertion/deletion in the Priority Queue (with the List Implementation)?
- the hash table (size) or hash function?
- How to use linear probing/Quadratic Probing/double hashing/separate chaining?
- Definition and Terminology of tree?
- The height of complete/full binary tree?
- How to calculate the slot index of child/parent nodes in binary tree?
- Given inorder/postorder traversal, the preorder traversal?
- Given a tree, the inorder/postorder/preorder traversal?
- Implementation of operations in binary tree (count_leaf/equal/height/…)?
- What is a Heap?
- Insertion/deletion in a heap?