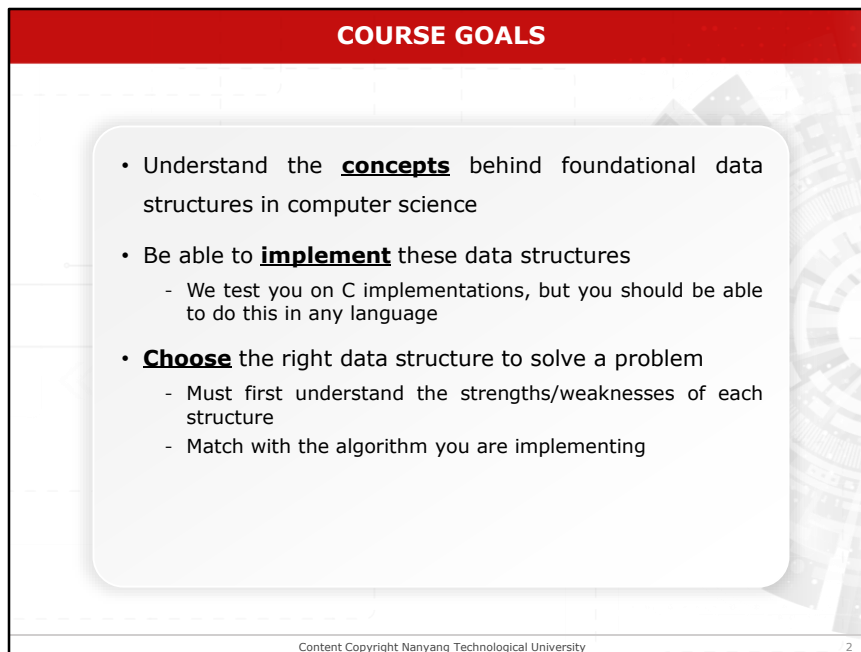


This lecture is on Data Structures Summary



## COURSE GOALS

- Understand the **concepts** behind foundational data structures in computer science
- Be able to **implement** these data structures
  - We test you on C implementations, but you should be able to do this in any language
- **Choose** the right data structure to solve a problem
  - Must first understand the strengths/weaknesses of each structure
  - Match with the algorithm you are implementing

Content Copyright Nanyang Technological University 2

At the end of this course, you should be able to implement data structures such as stacks, linked list, queues, trees, etc.

**DATA STRUCTURE COVERAGE**

- Data structures you must know (**concepts** and **implementation**) and may be tested on
  - **Linked lists**
  - **Stacks**
  - **Queues**
  - **Binary trees**
  - **Binary search trees**
  - Tree Balancing (**not required/tested**)
- Graph is **not** required/tested

Content Copyright Nanyang Technological University 3

Whether you should write the code for functions such as insertnode, removenode, etc. or you can quote those functions, is depend on the exam question.

Therefore practicing those basic functions are very important

## OVERVIEW

- For each data structure
  - Know the **basic concept**
  - Know how to implement in C
    - **Array based**
    - **Linked list based: Pointers + structures**
    - **Dynamic memory allocation/deallocation**
    - Code **reuse**: some structures implemented on top of other structures
  - Know **pros/cons** of each data structure
    - So that you can choose appropriate data structure for a problem
- Across data structures
  - Be able to **compare** and explain which is a better choice for a given task

### CONCEPT VS. IMPLEMENTATION

- Must be able to explain what a data structure is **without** referring to implementation details
  - Without talking about C structs or pointers
    - Some languages do not support structs or pointers
  - Many different ways to implement each data structure
- Explain how to use the concept behind each data structure to solve a problem

Content Copyright Nanyang Technological University 5

#### •Stack

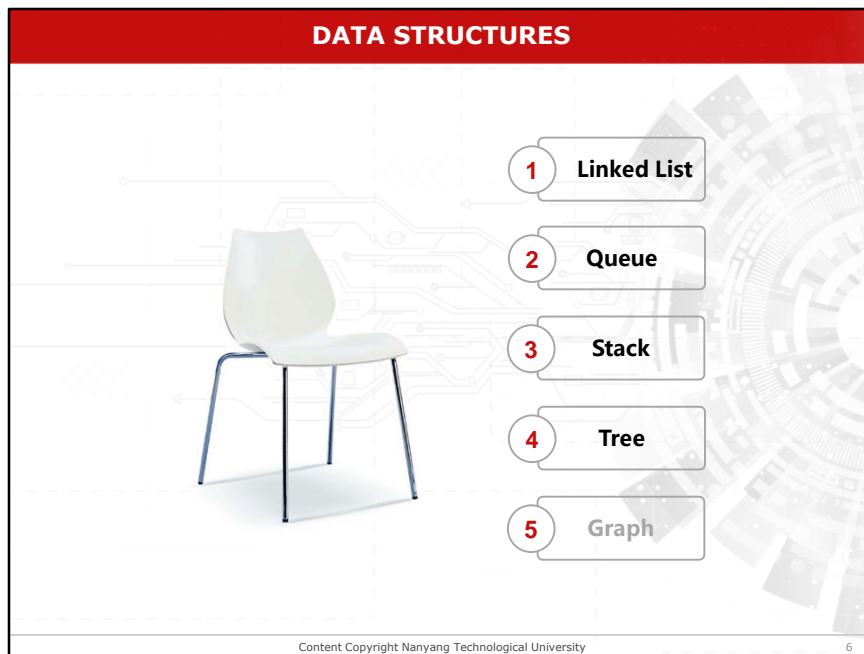
-Limited-access sequential data structures

-Stack: Last In, First Out (LIFO) principle

-Elements can only be added or removed at the top

-Deep relationship with recursion, backtracking

-Implement based on linked list or array




- What will we be working with? – Structures
  - Pointers
  - Structures inside structures
  - Pointers to structures
  - Pointers inside structures
- Make sure you know – What pointers/structures are – How to declare and use pointers/structures
- Draw lots of pictures
  - Visualizing how objects are laid out in memory helps with understanding
- Concept before code
  - Following pointers can be tricky if you don't have a mental model of the data structure
  - With the right model as a reference, you can implement the structure in any language
- Use the debugger

– Once you start writing code, you'll do silly things with pointers and you need to be able to track down your mistakes

## LINKED LISTS

- What is a linked list?
  - Ordered list of items
  - Each item stored in a node
  - Each node connects to the next node in the series
- No need for pointers in definition of a linked list
  - Head pointer, next pointer: all implementation details



Content Copyright Nanyang Technological University

7

In the definition of Linked List , pointer is not necessary.

Because when you use other languages to implement Linked list, those languages do not have the concepts of pointers, yet they can implement the connection between nodes.

Even in C language when we use array based Linked List we don't need the concept of pointers.



### MEMORY ALLOCATION IN C

- When you write program you may not know how much space you will need. C provides a mechanism called a heap, for allocating storage at run-time.
- The function **malloc** is used to allocate a new area of memory. If memory is available, a pointer to the start of an area of memory of the required size is returned otherwise **NULL** is returned.
- When memory is no longer needed you may free it by calling **free** function.
- The call to **malloc** determines size of storage required to hold **int** or the **float**.

Content Copyright Nanyang Technological University

8

Stack is used for static memory allocation and Heap for dynamic memory allocation.

Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and it's allocation is dealt with when the program is compiled.

Variables allocated on the heap have their memory allocated at run time and accessing this memory is a bit slower, but the heap size is only limited by the size of virtual memory.

A dynamic memory allocation uses functions such as malloc() to get memory dynamically.

It is necessary to free the allocated memory so that the memory can be reused. The free() function frees up (deallocates) memory that was previously allocated with malloc().

## NODES

- Node-based data structures
  - Nodes + connections between nodes
- Data structure size is not fixed
  - Can create a node at any point while the program is running
  - Dynamic memory allocation **malloc()**: `malloc(sizeof(...))`
  - Deallocation of dynamic memory **free()**
  - **Common mistakes: memory leak, buffer overflow**
- Pointers vs nodes
  - Pointers create connections between nodes
  - Pointers are not nodes

Content Copyright Nanyang Technological University

Array based implementation has some limitation.

## IMPLEMENTATION OF NODE

- Implementation details differ across languages
- But same fields will always be there:
  - Data
  - Connection(s) to other node(s)
- In C, ListNode is a C struct with several fields
  - item: this is a data type holding the data stored in the node
  - next: this is a pointer storing the address of the next node in the sequence

```
typedef struct _listnode{  
    int item;  
    struct _listnode *next;  
}ListNode;
```

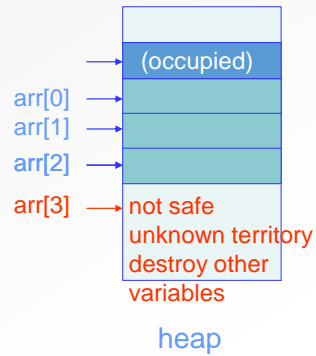
**MINIMUM  
SETTINGS**

## BUFFER OVERFLOWS

- Question: I used `malloc(3 * sizeof(int))` to allocate space for an array of 5 integers and it works. Why?

**Answer:**

- You have overwritten parts of memory that you were not supposed to
- These parts might store other variables or other program instructions
- Most of the time, this will crash your program
- But it might work if you are lucky



## MEMORY LEAKS

- When you allocate memory and then make it inaccessible, you have a memory leak
- This is very Bad.

```
#include <stdlib.h>
void main(){
    int *i;
    i = malloc(sizeof(int));
    i = malloc(sizeof(int));
}
```

After `i=malloc(sizeof(int))` is called the second time, no one is pointing to this block of memory

heap

Content Copyright Nanyang Technological University 12

Another common mistake can be identified as '**Memory Leaks**'.

**"When you allocate memory and then make it inaccessible, you have a memory leak"**

For an example, if you run the given code;

The first '`i=malloc(sizeof(int));`' code line will command the OS to allocate a memory block for integer `i`.

Now, `i` is pointing to the second memory block allocated, therefore the first allocated memory block is free because no one is pointing at it, yet it is inaccessible.

## LINKED LIST FUNCTIONS USING ListNode STRUCT

- Function prototypes:
  - **void printList(ListNode \*head);**
  - **ListNode \* findNode(ListNode \*head, int index);**
  - **int insertNode(ListNode \*\*ptrHead, int index, int value);**
  - **int removeNode(ListNode \*\*ptrHead, int index);**

Content Copyright Nanyang Technological University

13

- Solution for the identified problem is to define another C structure called 'LinkedList' and encapsulate all the elements which required to implement the linked list.
- Then we consider both variables as one pointer.

## COMMON MISTAKES

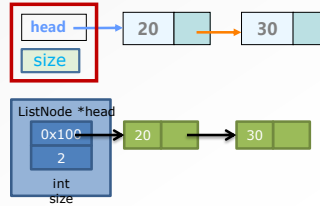
- **Forget** to check whether the list is empty  
**head=NULL**
- **Forget** to deal with the first node differently
- **Forget** to deal with the last node differently
- **Forget** to handle differently when: insert/remove a node at the beginning/tail of the list

## LINKEDLIST C STRUCT

- Implementation of Linked List

- Define another C struct, LinkedList
- Wrap up all elements that are required to implement the Linked List data structure

```
typedef struct _linkedList{
    ListNode *head;
    int size;
} LinkedList;
```



- Why is this useful?

Consider the rewritten Linked List functions

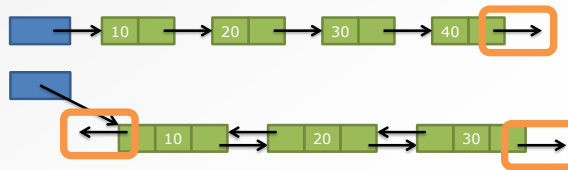


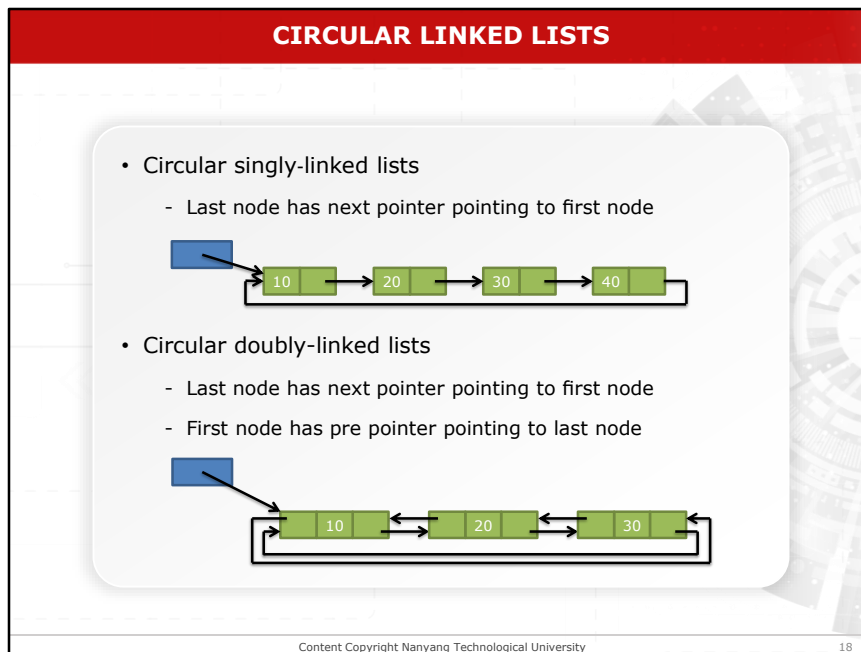
## LINKED LIST FUNCTIONS USING LinkedList STRUCT

- Original function prototypes:
  - void printList(ListNode \*head);
  - ListNode \* findNode(ListNode \*head, int index);
  - int insertNode(ListNode \*\*ptrHead, int index, int value);
  - int removeNode(ListNode \*\*ptrHead, int index);
- New function prototypes:
  - **void printList(LinkedList \*ll);**
  - **ListNode \* findNode(LinkedList \*ll, int index);**
  - **int insertNode(LinkedList \*ll, int index, int value);**
  - **int removeNode(LinkedList \*ll, int index);**

## MORE COMPLEX LINKED LISTS

- Singly-linked lists
  - So far, linked list has a fixed end
  - No way to loop around
- Doubly-linked lists
  - Might be useful to allow looping traversal
  - No extra variables needed in the ListNode struct
    - Just have to add connections

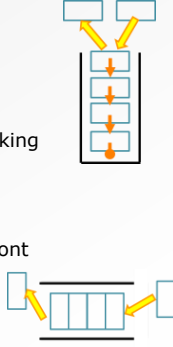




We can loop the linked list as a circle which then form a circular linked list.

## STACKS AND QUEUES

- What is a stack?
  - Ordered list of items
  - Add and remove only at the top
  - **Last In First Out**
  - Deep relationship with recursion, backtracking
- What is a queue?
  - Ordered list of items
  - Add at the back and remove only at the front
  - **First In First Out**
- How do we build stacks/queues?
  - Built on top of LinkedLists, arrays, etc.
  - These are all implementation issues



Content Copyright Nanyang Technological University 19

Because of the Stacks LIFO (Last In First Out) property it remembers its 'caller' and thus knows whom to return when the function has to return.

Recursion makes use of system stack for storing the return addresses of the function calls.

Every recursive function has its equivalent iterative (non-recursive) function.

Even when such equivalent iterative procedures are written, explicit stack is to be used.

## STACK DATA STRUCTURE

- A Stack is a data structure that operates like a physical stack of things
  - Stack of books, for example
  - Elements can only be added or removed at the top
- Key: Last-In, First-Out (LIFO) principle
  - Or First-In, Last-Out (FILO)
- Built on top of one other data structure
  - Arrays, linked lists, etc.
  - We'll focus on a linked list-based implementation



## STACK DATA STRUCTURE

- Core operations
  - Push: Add an item to the top of the stack
  - Pop: Remove an item from the top of the stack
- Common helpful operations
  - Peek: Inspect the item at the top of the stack without removing it
  - IsEmptyStack: Check if the stack has no more items remaining
- Corresponding functions
  - **push()**
  - **pop()**
  - **peek()**
  - **isEmptyStack()**
- We'll build a stack assuming that it only deals with integers
  - But as with linked lists, can deal with any contents depending on how you define the functions and the underlying implementation

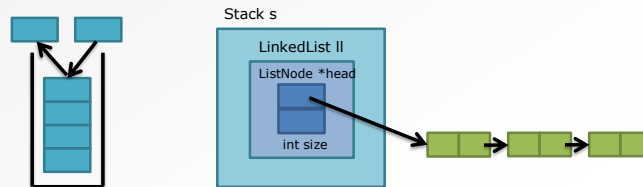
## STACK IMPLEMENTATION USING LINKED LISTS

- Stack structure

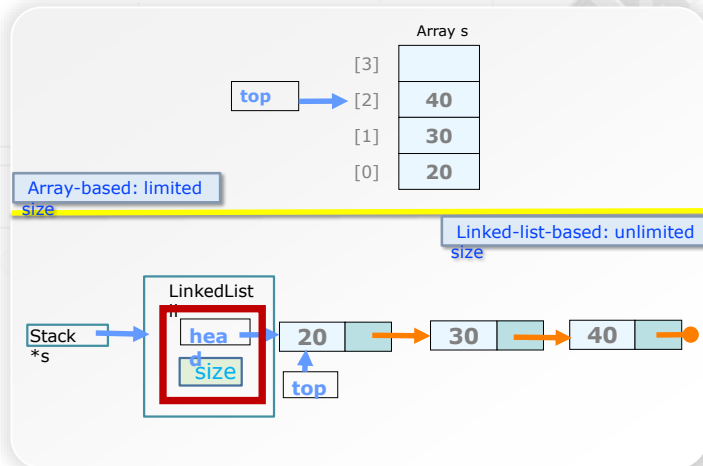
```
typedef struct _stack{
    LinkedList l1;
} Stack;
```

Notice this is a LinkedList, not a LinkedList \*

- Basically wrap up a linked list and use it for the actual data storage
- Just need to ensure we control where elements are added/removed
- Notice that the LinkedList already takes care of little things like keeping track of number of nodes, etc.



## STACK: ARRAY-BASED VS. LINKED-LIST-BASED





## QUEUE DATA STRUCTURE

- A Queue is a data structure that operates like a real-world queue
  - Elements can only be added at the back
  - Elements can only be removed from the front
- Key: **First-In, First-Out (FIFO) principle**
  - Or, Last-In, Last-Out (LILO)
- Often built on top of some other data structure
  - Arrays, Linked lists, etc.
  - We'll focus on a linked list-based implementation



## QUEUE DATA STRUCTURE

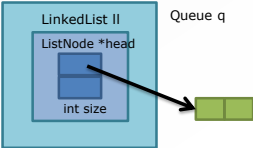
- Core operations
  - Enqueue: Add an item to the back of the queue
  - Dequeue: Remove an item from the front of the queue
- Common helpful operations
  - Peek: Inspect the item at the front of the queue without removing it
  - IsEmptyStack: Check if the queue has no more items remaining
- Corresponding functions
  - **enqueue()**
  - **dequeue()**
  - **peek()**
  - **isEmptyQueue()**
- We'll build a queue assuming that it only deals with integers
  - But as with linked lists, can deal with any contents depending on your code

## QUEUE IMPLEMENTATION USING LINKED LISTS

- Recall that we defined a LinkedList structure
 

```
typedef struct _linkedlist{
    ListNode *head;
    int size;
}LinkedList;
```
- Now, define a Queue structure
  - We'll build our queue on top of a linked list

```
typedef struct _queue{
    LinkedList ll;
}Queue;
```



The diagram illustrates the internal structure of a Queue. A light blue box labeled 'Queue q' contains a smaller light blue box labeled 'LinkedList ll'. Inside the 'LinkedList ll' box, there is a 'ListNode \*head' field and an 'int size' field. An arrow points from the 'head' field to a linked list structure consisting of two green rectangular nodes connected by a line.

Content Copyright Nanyang Technological University

26

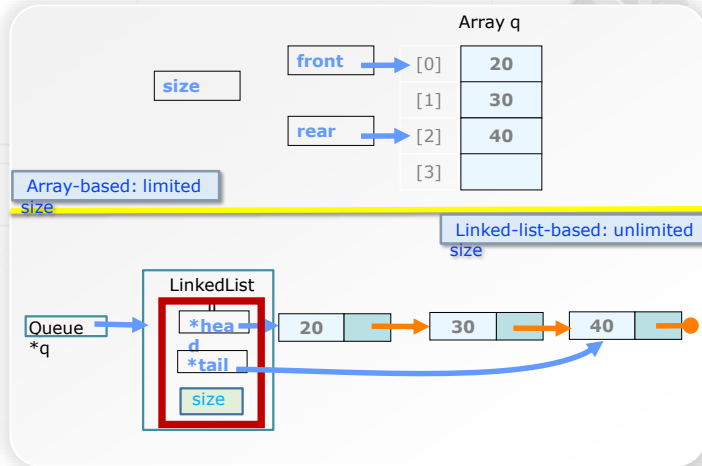
### Queue Structure

- Define the queue structure
 

```
type def struct – queue { ..... } Queue;
```
- Inside the definition, here we declare a variable which has linked list type
 

```
LinkedList ll;
```

## QUEUE: ARRAY-BASED VS. LINKED-LIST-BASED



## YOU SHOULD FIGURE OUT WHICH ONE IS POINTER, WHICH ONE IS NOT

Queue structure

```
typedef struct _queue{
    LinkedList ll;
}Queue;
```

When should I use "->" or "." ?  
 pointer->  
 Non-pointer.

Queue q;

q.ll

LinkedList

q.ll.head

ListNode pointer

q.ll.head->num

integer

Queue \*q;

q->ll

Is a pointer  
 LinkedList

q->ll.head

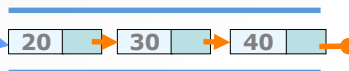
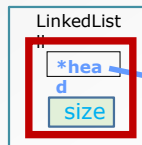
ListNode

q->ll.head->num

integer

Queue  
q

Queue\* q



**QUEUE AND STACK IMPLEMENTATION USING LINKED LISTS**

```
typedef struct _listnode{
    int num;
    struct _listnode *next;
}ListNode;

typedef struct _linkedlist{
    ListNode *head;
    int size;
}LinkedList;

typedef struct _stack {
    LinkedList ll;
}Stack;

typedef struct _listnode{
    int num;
    struct _listnode *next;
}ListNode;

typedef struct _linkedlist{
    ListNode *head;
    ListNode *tail;
    int size;
}LinkedList;

typedef struct _queue {
    LinkedList ll;
}Queue;
```

## BINARY TREES

- What is a binary tree?
  - Tree structure
    - Data structure that represents a hierarchical conceptual structure
  - At most two children per node
- Implementation of binary tree
  - In C, create a BTreeNode struct
    - item: Data field
    - left: Pointer to the left child node, NULL if none
    - right: Pointer to the right child node, NULL if none

## IMPLEMENTATION

- Recall implementation of LinkedList

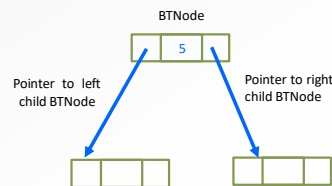
- Node has link to **at most one** other node
- Defined a ListNode with one **next** pointer and a data **item**

```
typedef struct _listnode{
    int item;
    struct _listnode *next;
}ListNode;
```

- BinaryTree

- A node has link to **at most TWO** other nodes
- Define a BTreeNode with
  - Two pointers
  - A data item

```
typedef struct _bnode{
    int item;
    struct _bnode *left;
    struct _bnode *right;
} BTreeNode;
```



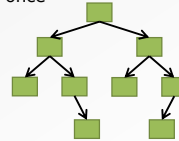
The C structure we have implemented for list node is;

The same method can be used for tree structure as well. But binary tree node has links to at most two other nodes. Therefore there should be two pointers.



## TREE OPERATIONS

- Recursive TreeTraversal
  - It guarantees every node will be visited exactly once
- Traversal orders
  - **Pre-order: C L R**
  - **In-order: L C R**
  - **Post-ordered: L R C**
- Without using recursion
  - Using a **queue: Breadth first** (level by level) traversal
  - Using a **stack: Iterative** pre-order traversal
- When writing your tree functions, consider the following
  - Does the **final answer propagate** down from the root or up from the leaves?
  - What information do I need **to pass to my children** when I visit them?
  - What information do I need **to pass to my parent** when I return?



Content Copyright Nanyang Technological University

32

- **Pre-order**
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - Visit the left child subtree
  - Process the current node's data
  - Visit the right child subtree
- **Post-order**
  - Visit the left child subtree
  - Visit the right child subtree
  - Process the current node's data

## TREE APPLICATIONS EXAMPLES

- **Count** nodes in a binary tree
- **Find grandchild** nodes
- **Height** of a node = number of links from that node to the deepest leaf node
- **Depth** of a node = number of links from that node to the root node

Content Copyright Nanyang Technological University

33

Count: Information propagates : **Pre, In, Post**

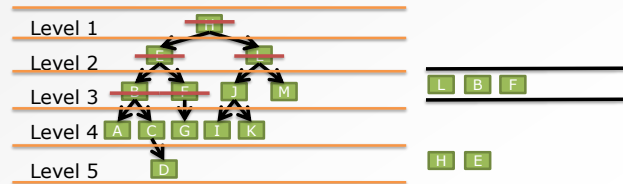
**Find grandchild** : information propagate **Downwards: Pre**

**Height** of a node : Information propagates **upwards: Post**

**Depth** of a node: information propagate **Downwards: Pre**

## LEVEL-BY-LEVEL TREE TRAVERSAL

- **Enqueue** the root, H
- **Dequeue** H, and **enqueue** H's children
- **Dequeue** E, and **enqueue** E's children



## QUEUE STRUCTURE

BSTNode



```
typedef struct _bstnode {
    int item;
    struct _bstnode *left;
    struct _bstnode *right;
} BSTNode;
```

QueueNode



```
typedef struct _QueueNode {
    BSTNode *data;
    struct _QueueNode *next;
} QueueNode;
```

Queue



```
typedef struct _queue {
    QueueNode *head;
    QueueNode *tail;
} Queue;;
```

## PREORDER TRAVERSAL WITH A STACK

**Push** the root onto the stack

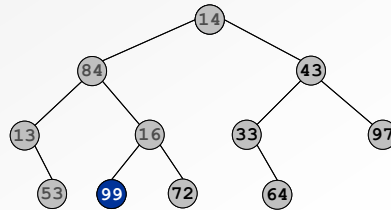
While the stack is not empty

- **pop** the stack and visit it
- **push** its two children

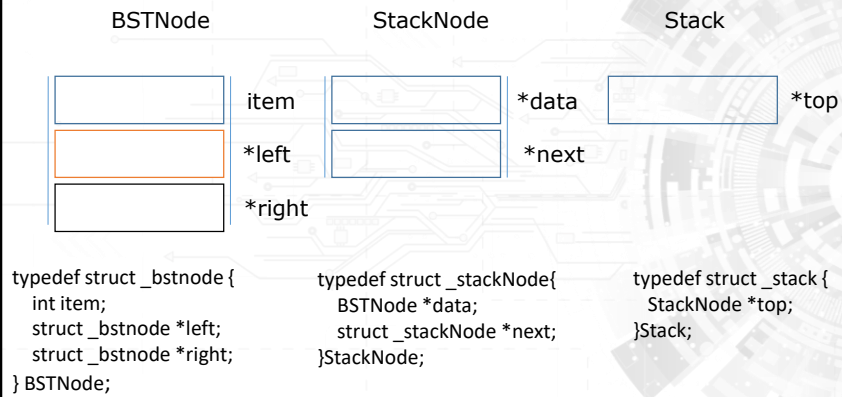
14 84 13 53 16 99

72  
43

Stack



## STACK STRUCTURE



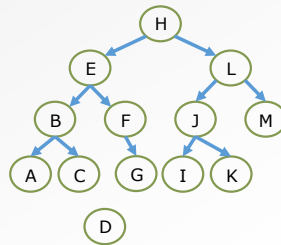
## BINARY SEARCH TREES

- What is a BST?
  - A BT where the **L < C < R** rule is enforced
    - Recursively,
      - **C** is the data in the current node
      - **L** represents the data in any/all nodes from C's left subtree
      - **R** represents the data in any/all nodes from C's right subtree
- BSTs allow for
  - **Efficient search**
  - Easy storage of a list of items in sorted order
    - **In-order traversal produces a sorted list**
    - Insertion in "sorted order" is also efficient

## INSERTING A NODE INTO A BST

- Key point:

- Given an existing BST and a new value to store, there is always a unique position for the new value
- Node insertion is relatively simple!



There is only one unique position a node can be inserted in. Therefore there is only one answer.

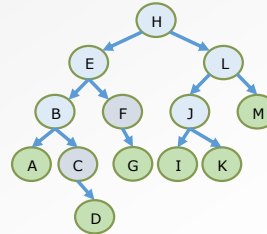


## REMOVING A NODE FROM A BST

- Remove node X - a bit tricky

- 3 cases:

1. x has no children:
  - Remove x
2. x has one child y:
  - Replace x with y
3. x has two children:
  - Swap x with successor
  - Perform case 1 or 2 to remove it



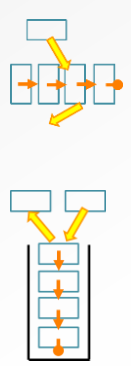
There are 3 cases we should consider when we try to remove a node from a BST.

**PICK A STRUCTURE**

- Linked lists vs stack
- Linked lists vs binary trees
- Linked lists vs binary search trees
- Stacks vs binary search trees
- Binary trees vs binary search trees

### LINKED LIST VS STACK

- Linked lists (&Array)
  - Can access and do operations to any item
- Stack
  - **Limited-access** sequential data structures
  - Stack: **Last In, First Out** (LIFO)
  - Implement based on linked list or array



Content Copyright Nanyang Technological University

42

It's possible to insert a node, or remove a node, from anywhere in a linked list.

A queue has pointers to both its head and its tail so that nodes may be inserted at the tail and deleted from the head.

## LINKED LISTS VS BINARY (SEARCH) TREES

- Linked list is for **linear** data
  - Each node has at most one link to other node
  - Simple traversal
- Binary Tree is for **hierarchical** data
  - Each node has at most two links to other nodes
  - Different order of traversals, more complicated than list
- For item search:
  - Binary search trees
    - Medium complexity to implement, expensive to maintain
    - Lookups are **efficient**, about the height of the tree
  - Linked lists (unsorted)
    - Low complexity to implement, easy to maintain
    - Lookups are **inefficient**, about the size of the list

## BINARY TREES VS BINARY SEARCH TREES

- A BST is a BT
- BST is **efficient** in item searching compared to normal BT.
- BST has the following features:
  - The left child only contains nodes with values less than the parent node;
  - The right child only contains nodes with values greater than the parent node;
  - There must be no duplicate nodes.



The slide features a red header bar with the text "CONTACT INFORMATION". Below this, a white rounded rectangle contains a bulleted list of contact details. The background of the slide is a light gray grid with a faint architectural image on the right side.

**CONTACT INFORMATION**

- Owen Noel **Newton Fernando**
- Email: **oferlando@ntu.edu.sg**
- Office: **NTU, N4-02c-80**
- Phone: **6908-3322**

Content Copyright Nanyang Technological University 45

Contact information