CE1007/CZ1007 DATA STRUCTURES

Lecture 12: **Data Structures Summary**

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

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

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

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

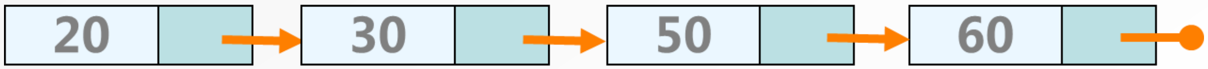
CONCEPT VS. IMPLEMENTATION

- 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

DATA STRUCTURES

1. Linked List
2. Queue
3. Stack
4. Tree
5. Graph

- 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



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***.

NODES

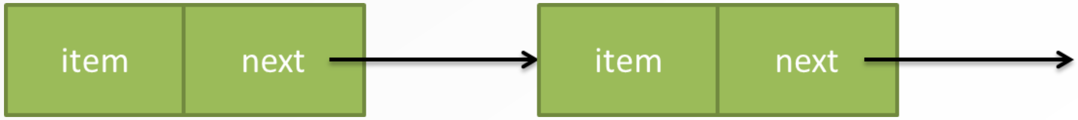
- Node-based data structures
  - Nodes + connections between nodes
- Data structure size is not fixed
  - Can create a node at any point while the program is running
  - Dynamic memory allocation **malloc()**: malloc(sizeof(…))
  - Deallocation of dynamic memory **free()**
  - **Common mistakes: memory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

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

MINIMUM SETTINGS

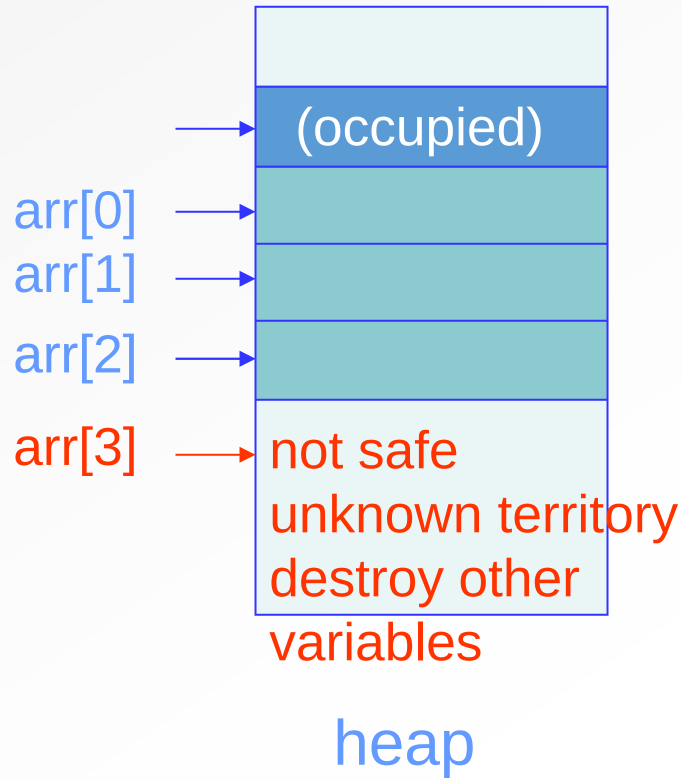
```
typedef struct _listnode{
   int item;
   struct _listnode *next;
}ListNode;
```



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

Answer:

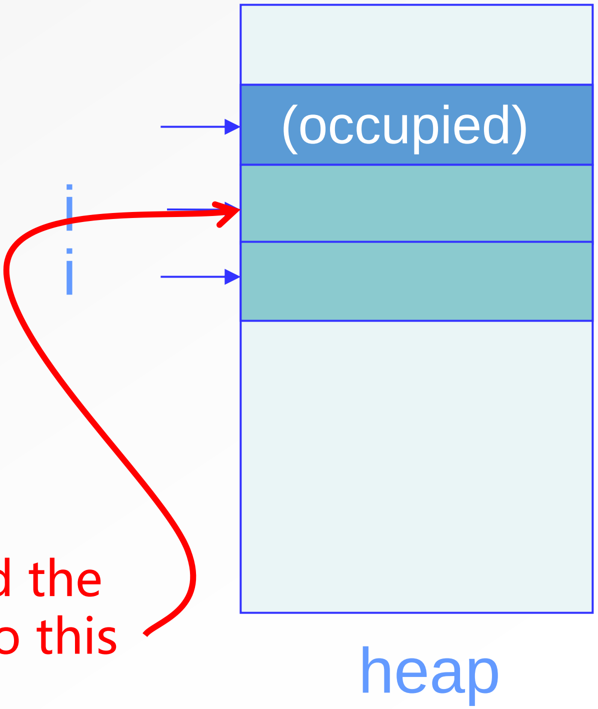
- **You have overwritten parts of memory that you were not supposed to**
- **These parts might store other variables or other program instructions**
- **Most of the time, this will crash your program**
- **But it might work if you are lucky**



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

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);**

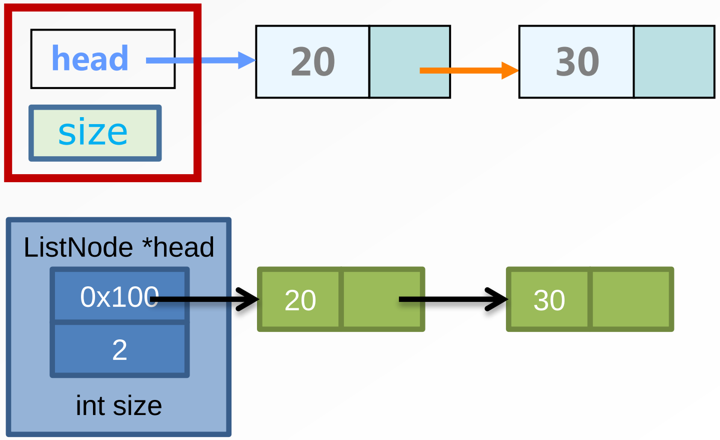
COMMON MISTAKES

- **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

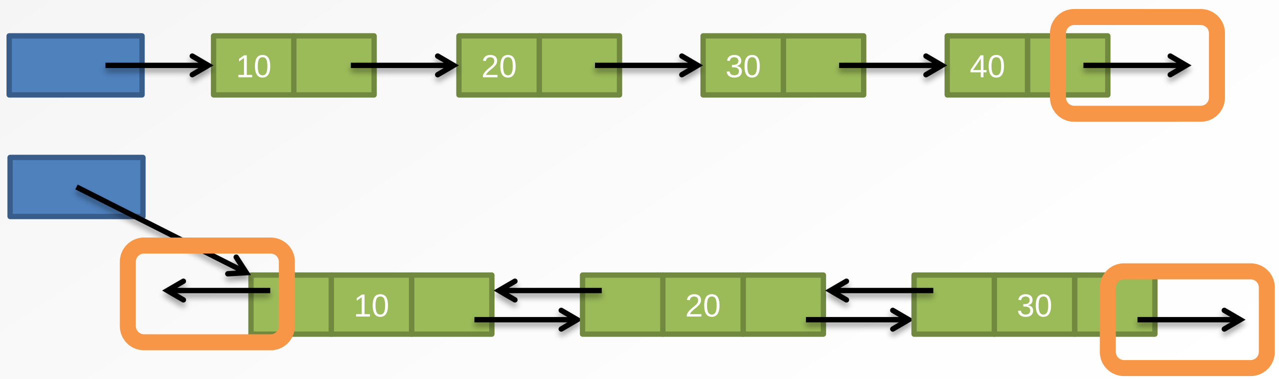
LINKED LIST FUNCTIONS USING LinkedList STRUCT

- 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);**

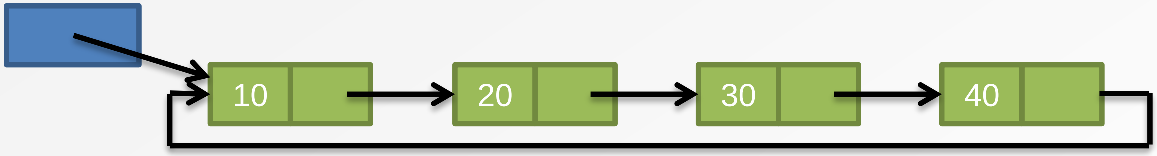
MORE COMPLEX LINKED LISTS

- 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

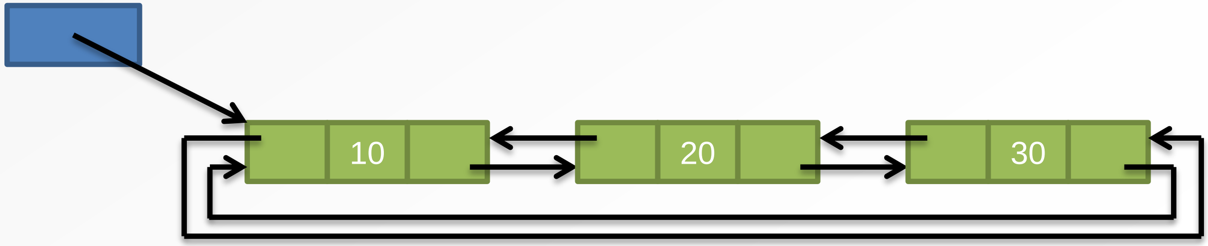


CIRCULAR LINKED LISTS

- Circular singly-linked lists
  - Last node has next pointer pointing to first node

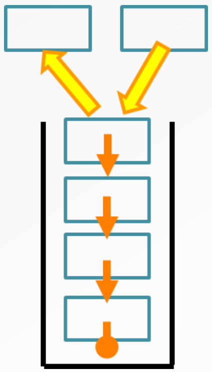


- Circular doubly-linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node

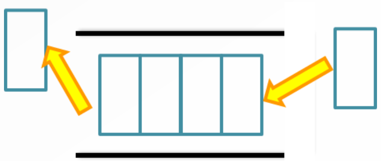


STACKS AND QUEUES

- 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 <u>implementation</u> issues

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

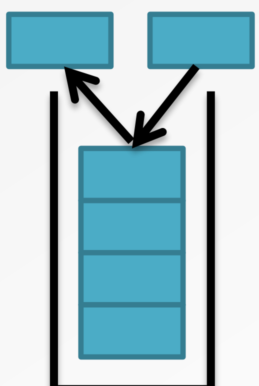
STACK IMPLEMENTATION USING LINKED LISTS

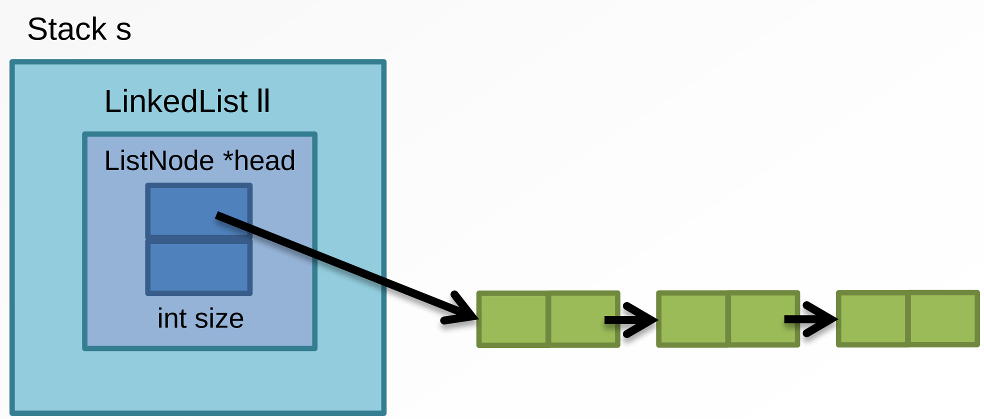
- Stack structure

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

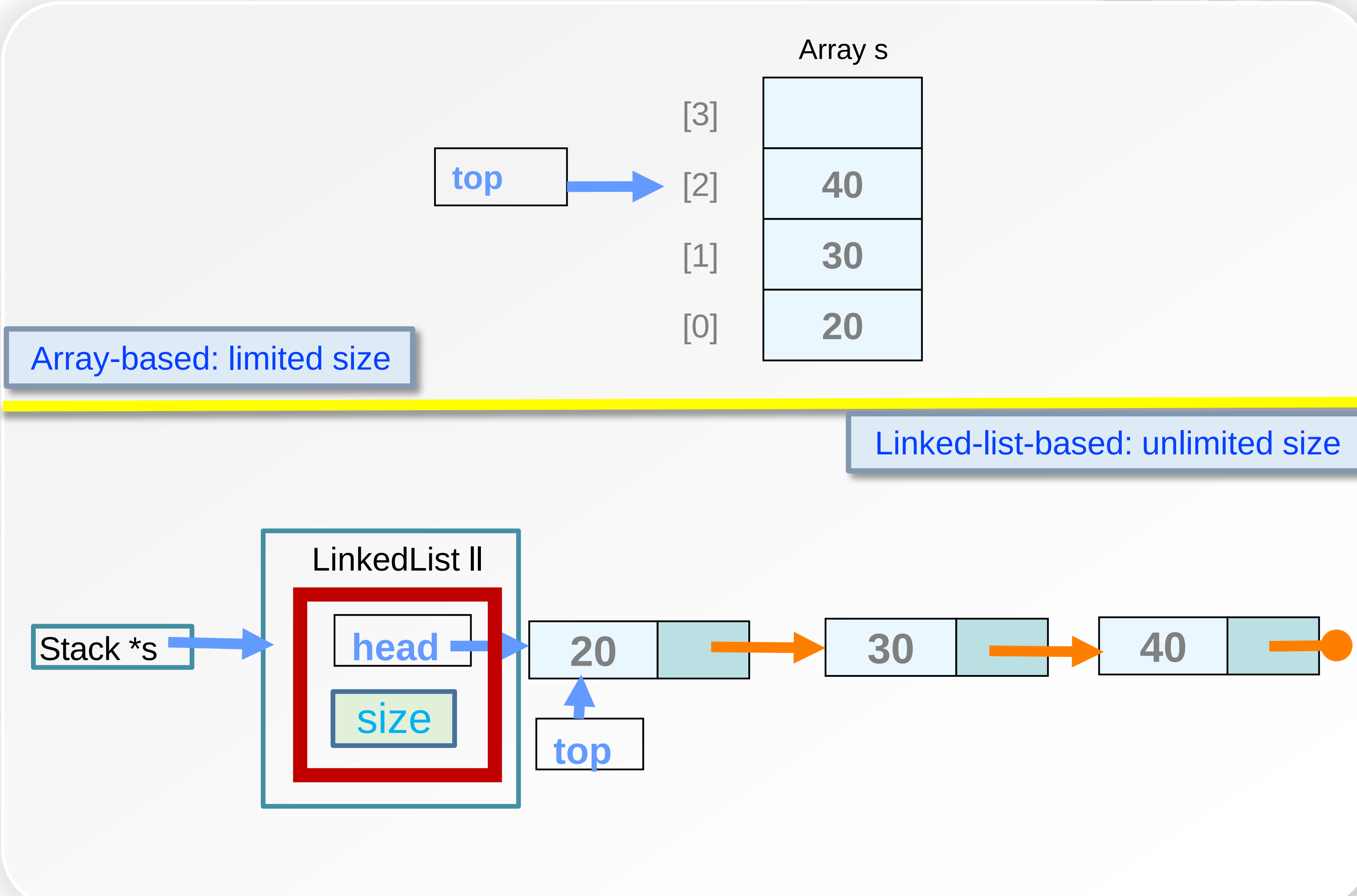
Notice this is a LinkedList, not a LinkedList *

- 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

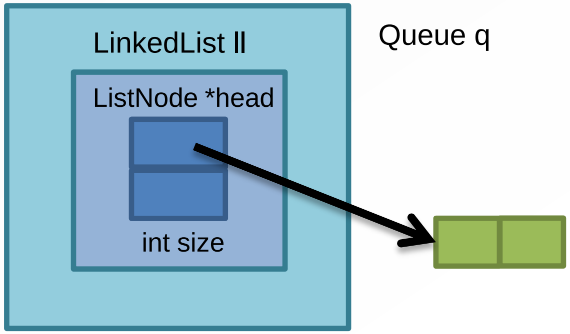
QUEUE IMPLEMENTATION USING LINKED LISTS

- Recall that we defined a LinkedList structure

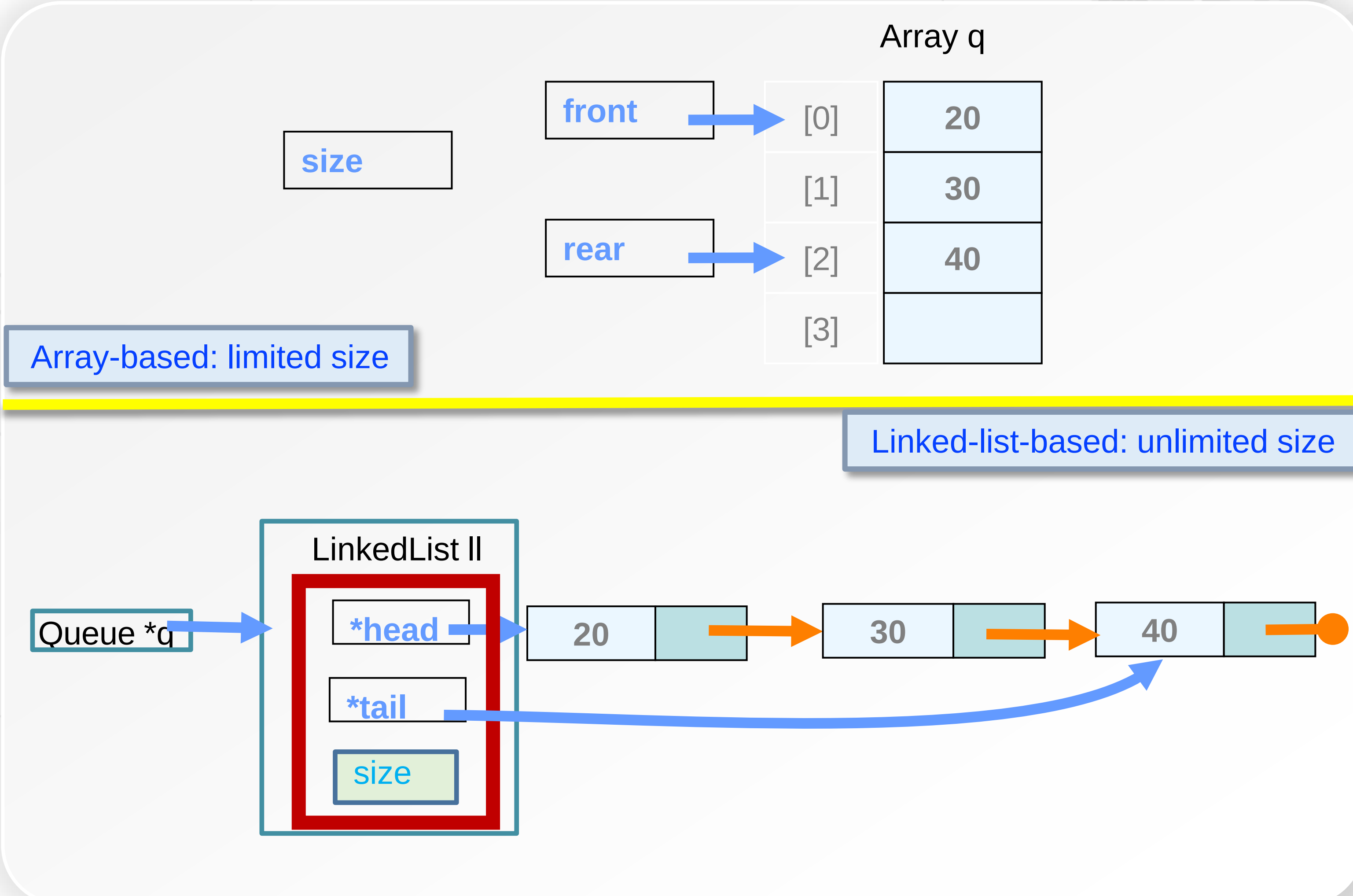
```
typedef struct _linkedlist{
      ListNode *head;
      int size;
}Linked List;
```

- Now, define a Queue structure
  - We'll build our queue on top of a linked list

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



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



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

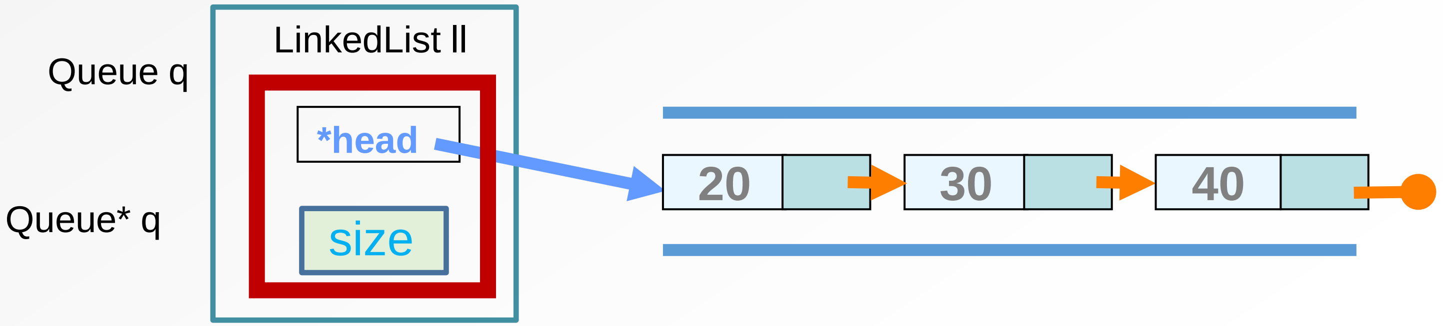
When should I use "->" or "." ?
pointer->
Non-pointer.

Queue structure

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

Queue q;
- **q.ll** — **LinkedList**
- **q.ll.head** — ListNode pointer
- **q.ll.head->num** — **integer**

Queue *q; — Is a pointer
- **q->ll** — **LinkedList**
- **q->ll.head** — ListNode pointer
- **q->ll.head->num** — **integer**



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 BTNode struct
    - item: Data field
    - left: Pointer to the left child node, NULL if none
    - right: Pointer to the right child node, NULL if none

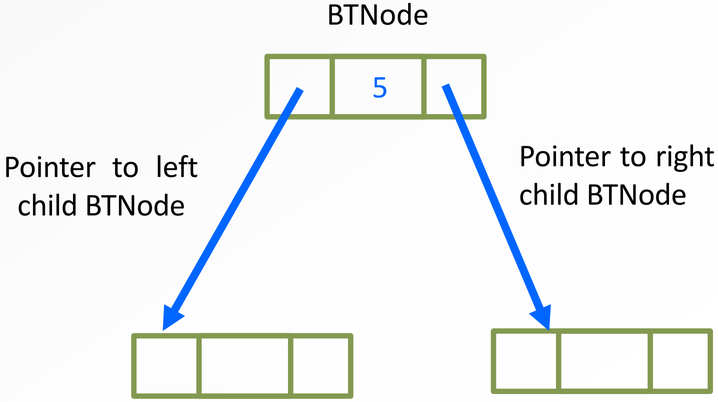
IMPLEMENTATION

- Recall implementation of LinkedList
  - Node has link to **at most <u>one</u>** other node
  - Defined a ListNode with one **`next`** pointer and a data **`item`**

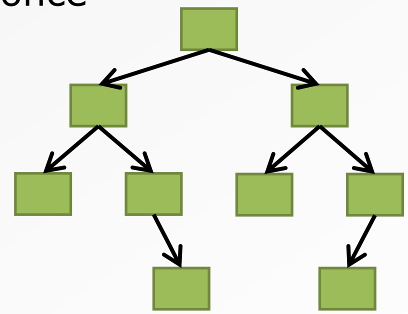
```
typedef struct _listnode{
    int item;
    struct _listnode *next;
}ListNode;
```

- BinaryTree
  - A node has link to **at most <u>TWO</u>** other nodes
  - Define a BTNode with
    - Two pointers
    - A data item

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```



- Recursive TreeTraversal
  - It guarantees every node will be visited exactly once

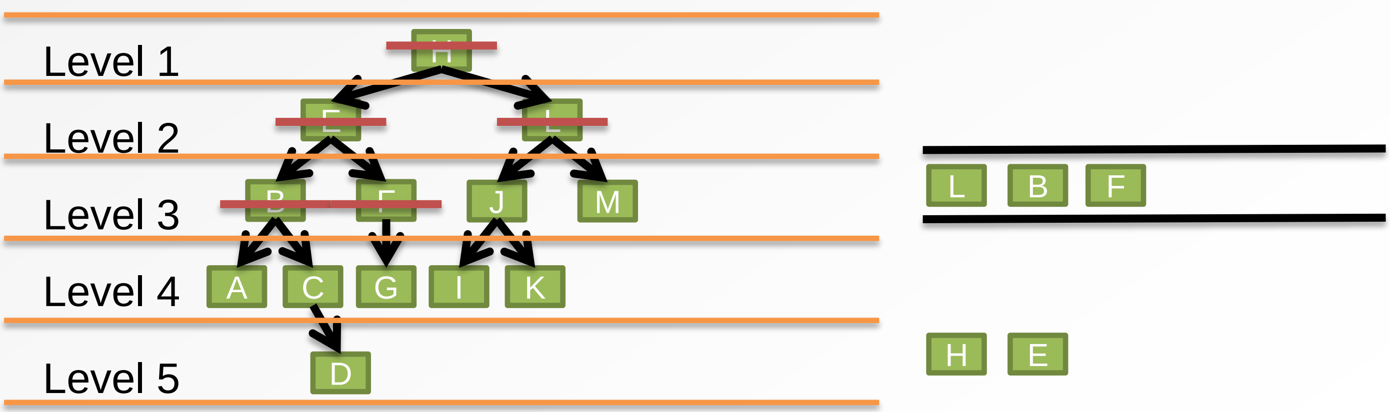


- Traversal orders
  - **Pre-order: C L R**
  - **In-order: L C R**
  - **Post-orded: L R C**
- Without using recursion
  - Using a **queue**: **Breadth first** (level by level) traversal
  - Using a **stack**: **Iterative** pre-order traversal
- When writing your tree functions, consider the following
  - 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?

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

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

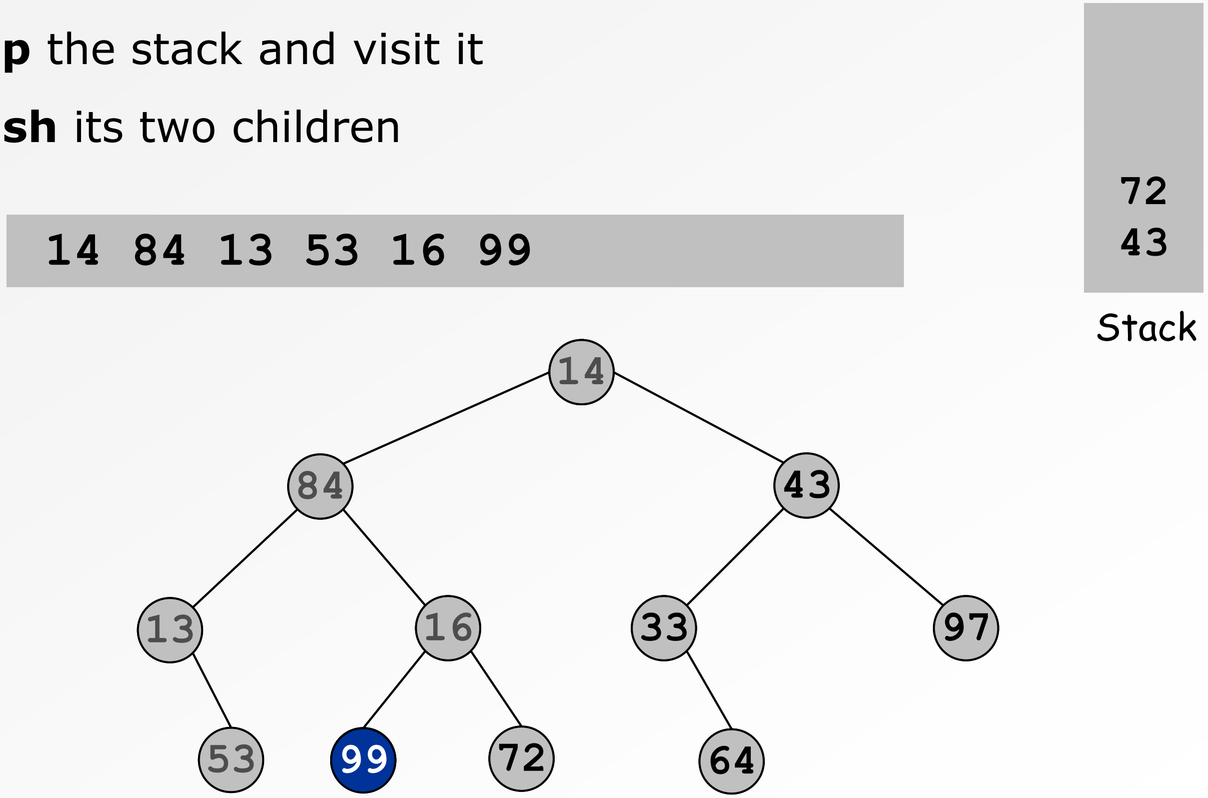


PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack

While the stack is not empty

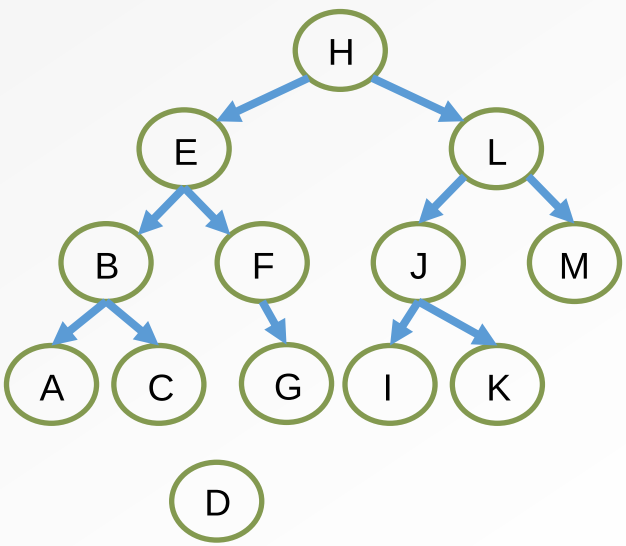
- **pop** the stack and visit it
- **push** its two children



- 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

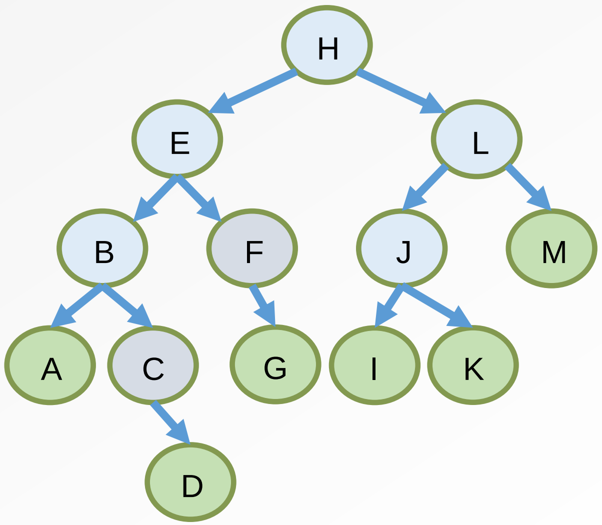
INSERTING A NODE INTO A BST

- 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!



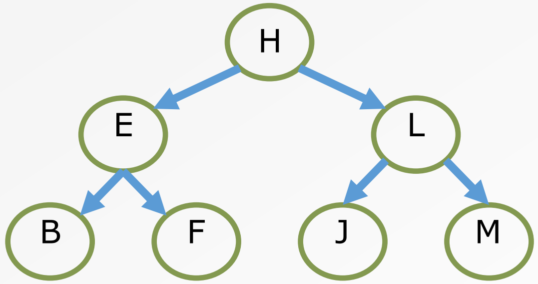
REMOVING A NODE FROM A BST

- 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

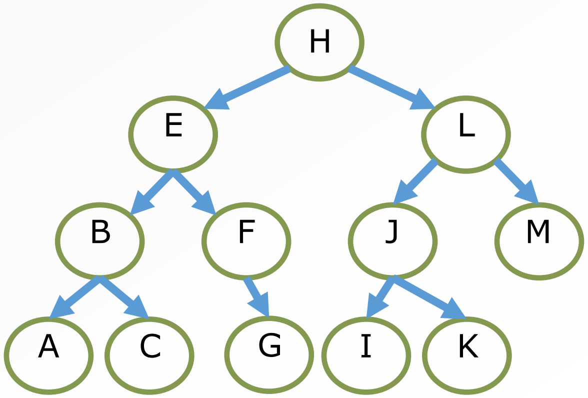


TREE BALANCING

- Goal: BST with the shortest height (short external paths, most efficient search)
- Ideal BST: Shortest height, **$H = \lfloor \log_2 n \rfloor$**
- The maximum number of nodes possible in a BST of height **H** is **$2^{(H+1)} - 1$**



$n = 2^{(H+1)} - 1 = 2^{2+1} - 1$
$n = 7$ $H=2$



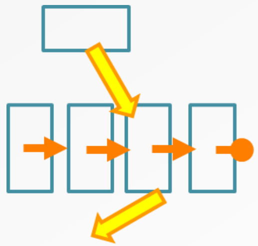
$n = 12$ $H=3$
$n < 2^{3+1} - 1 = 15$

PICK A STRUCTURE

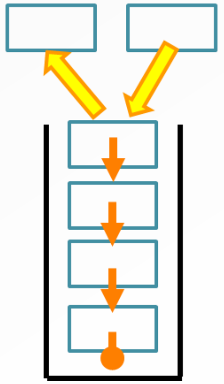
- Linked lists vs stack
- Linked lists vs binary trees
- Linked lists vs binary search trees
- Stacks vs binary search trees
- Binary trees vs binary search trees

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



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.

CONTACT INFORMATION

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