NANYANG TECHNOLOGICAL UNIVERSITY

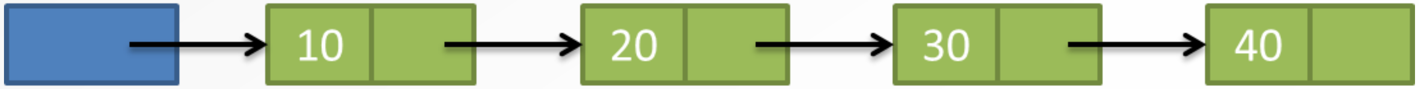
Summary: Linked List

College of Engineering

School of Computer Science and Engineering

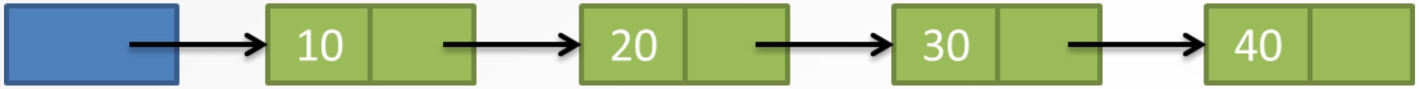
LINKED LISTS

- 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 <u>implementation</u> details



BASIC LINKED LIST

- Different types of data can be stored in a node
- Singly-linked list
    - Each node is connected to at most one other node
    - Each node keeps track of the next node



NODES

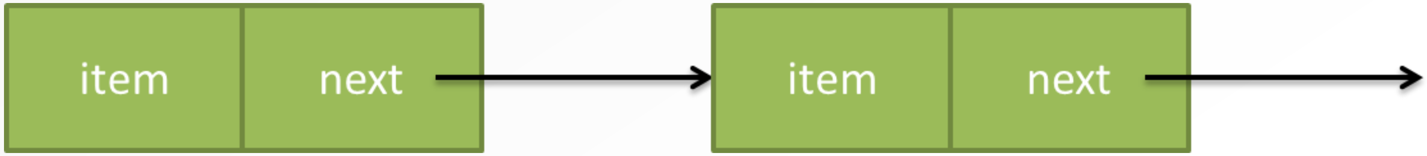
- Node-based data structures
  - Nodes + connections between nodes
- Data structure size is not fixed
  - Can create a node at any point while the program is running
  - Dynamic memory allocation malloc(): malloc(sizeof(…))
  - Deallocation of dynamic memory free()
  - **Common mistakes: memory leak,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

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

MINIMUM SETTINGS

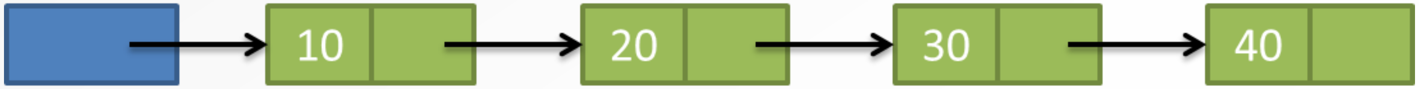


Content Copyright Nanyang Technological University

LINKED LIST FUNCTIONS USING LISTNODE STRUCT

Function prototypes:

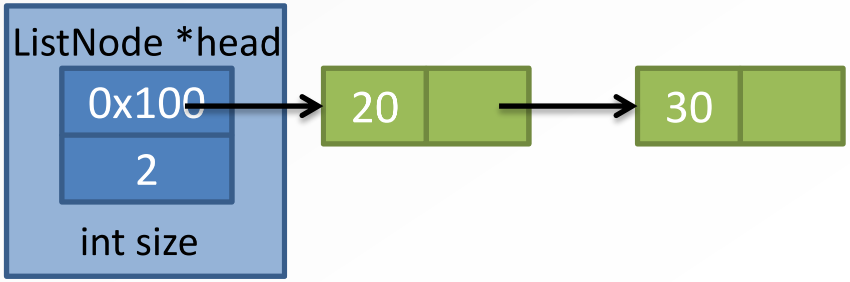
- `void printList(ListNode *head);`
- `ListNode * findNode(ListNode *head, int index);`
- `int insertNode(ListNode **ptrHead, int index, int value);`
- `int removeNode(ListNode **ptrHead, int index);`



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

Content Copyright Nanyang Technological University

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);
```

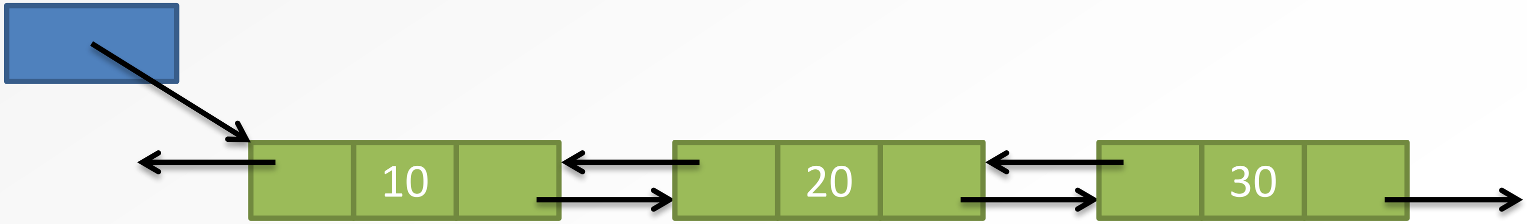
- New function prototypes:

```
– void printList(LinkedList *ll);
– ListNode * findNode(LinkedList *ll, int index);
– int insertNode(LinkedList *ll, int index, int value);
– int removeNode(LinkedList *ll, int index);
```

Content Copyright Nanyang Technological University

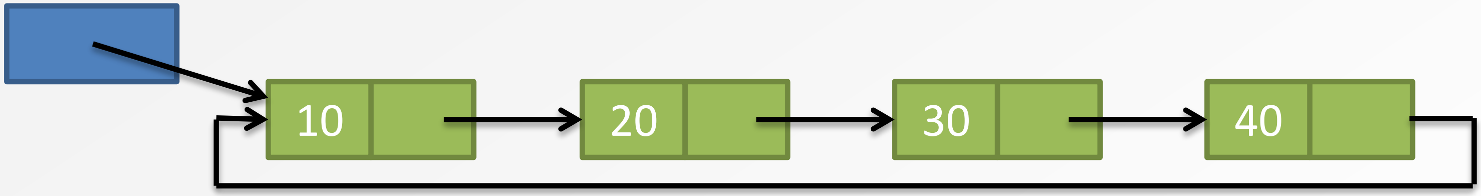
MORE COMPLEX LINKED LISTS - DOUBLY LINKED LIST

- So far, singly-linked list
  - Each ListNode is linked to at most one other ListNode
  - Traversal of the list is one-way only
    - Can’t go backwards
    - What if we want to start from a given node and search EITHER backwards OR forwards
- Doubly Linked List
  - Traversing a doubly linked list in forward direction
  - Traversing a doubly linked list in backward direction

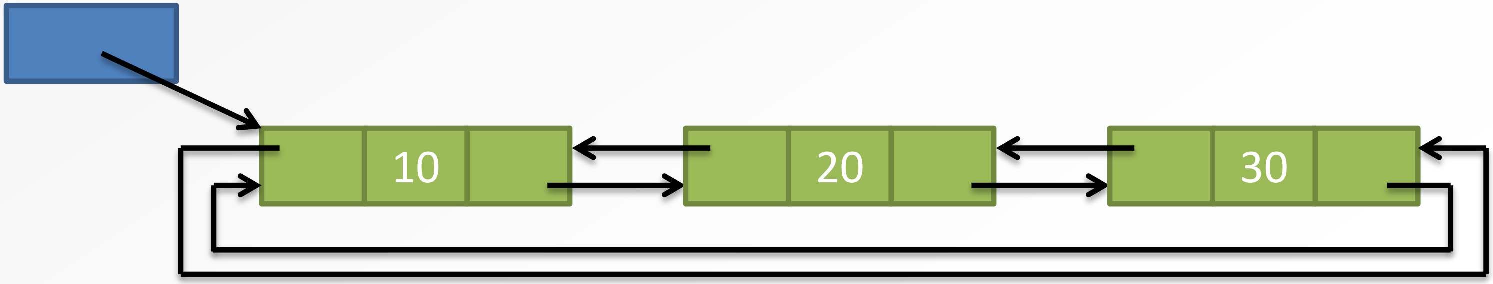


MORE COMPLEX LINKED LISTS - CIRCULAR LINKED LISTS

- 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 nod



ARRAYS VS. LINKED LISTS

- **Arrays**
  - Efficient random access
  - Difficult to expand, re-arrange
  - When inserting/removing items in the middle or at the front, computation time scales with size of list
  - Generally a better choice when data is immutable
- **Linked lists (dynamic-pointer-based and static-array-based)**
  - “Random access” can be implemented, but more inefficient than arrays
  - cost of storing links, only use internally.
  - Easy to shrink, rearrange and expand (but array-based linked list has a fixed size)
  - Insert/remove operations only require fixed number of operations regardless of list size. no shifting

Content Copyright Nanyang Technological University

COMMON MISTAKES

- Very important!
  - head is a node pointer
  - Points to the first node
  - head is not the “first node”
  - head is not the “head node”
- 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
- Changes of the links when insert/remove a node. The order matters!!

Content Copyright Nanyang Technological University