NANYANG TECHNOLOGICAL UNIVERSITY

CE1007/CZ1007 DATA STRUCTURES

Face-to-Face Session 4 & 5

Advanced Linked List, Stack and Queue

Dr. LOKE Yuan Ren

yrloke@ntu.edu.sg

N4-02b-69a

College of Engineering

School of Computer Science and Engineering

SO FAR...

Dynamic Memory Management

- `#include <stdlib.h>`
- `malloc()` dynamically memory allocation.
- `free()` deallocate memory

Linked List

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

Interface Functions

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`

LINKED LIST VS ARRAY

1. **Display: Both are similar**
2. **Search: Array is better**
3. **Insert and Delete: Linked List is more flexible**
4. **Size: Array is better**

Can we improve our sizeList()?

Interface Functions

1. Display: printList()
2. Search: findNode()
3. Insert: insertNode()
4. Delete: removeNode()
5. Size: sizeList()

…

```c
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```

```c
int sizeList(ListNode *head){
    int count = 0;
    while (head != NULL){
            count++;
            head = head->next;
    }
    return count;
}
```

```c
ListNode *findNode(ListNode* cur, int i){
    if (cur==NULL || i<0)
        return NULL;
    while(i>0){
        cur=cur->next;
        if (cur==NULL)
            return NULL;
        i--;
    }
    return cur;
}
```

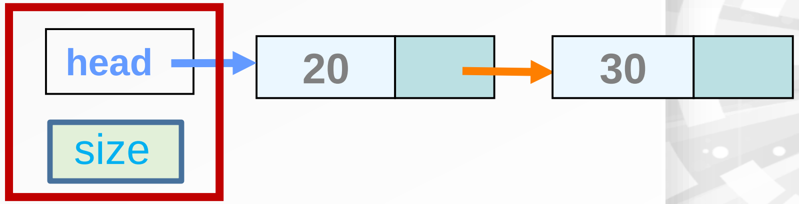
```c
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```

CAN WE IMPROVE OUR sizeList()?

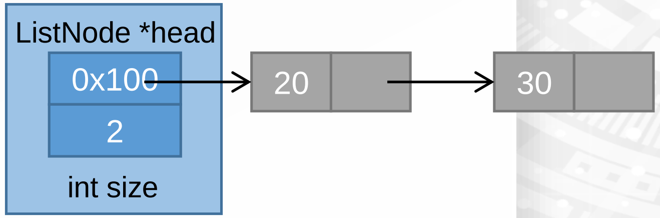
```c
int sizeList(ListNode *head){
    int count = 0;
    while (head != NULL){
            count++;
            head = head->next;
    }
    return count;
}
```

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

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




```c
int sizeList(LinkedList ll){
    return ll.size;
}
```



- Remember to change size when adding/removing nodes

LINKED LIST FUNCTIONS USING LinkedList STRUCT

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

NEW findNode()

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

```
ListNode *findNode(ListNode* cur, int i){
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```

```
ListNode *findNode(LinkedList ll, int i){
    ListNode *temp = ll.head;
    if (cur==NULL || i < 0|| i >ll.size)
        return NULL;

    while (i > 0){
        temp = temp->next;
        if (temp == NULL)
            return NULL;
        i--;
    }
    return temp;
}
```

1. Variations of the Linked List
    - Doubly-linked Lists
    - Circular Linked Lists
    - Circular Doubly-linked Lists
2. Stack
3. Queue

Variations of the Linked List

- Doubly-linked Lists
- Circular Linked Lists
- Circular Doubly-linked Lists

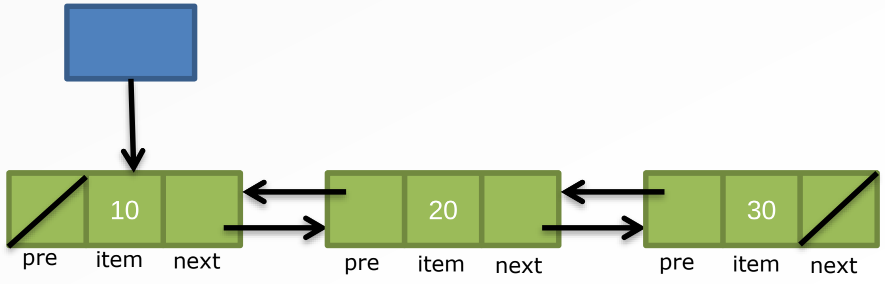
DOUBLY LINKED LIST

- Singly Linked list: Only one link. Traversal of the list is one way only.

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

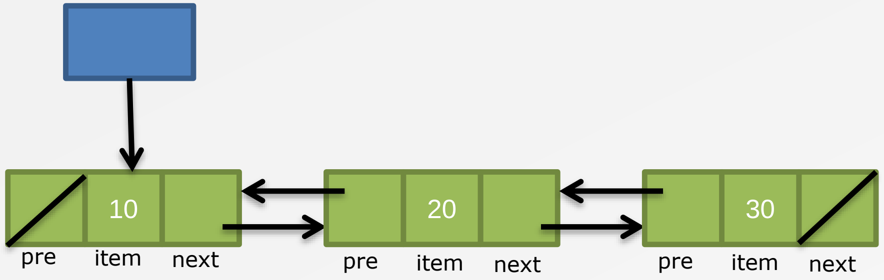
- Doubly Linked List: two links in each node. It can search forward and backward

```
struct _dbllistnode
{
    int item;
    struct _dbllistnode *pre;
    struct _dbllistnode *next;
};
typedef struct _dbllistnode DblListNode;
```



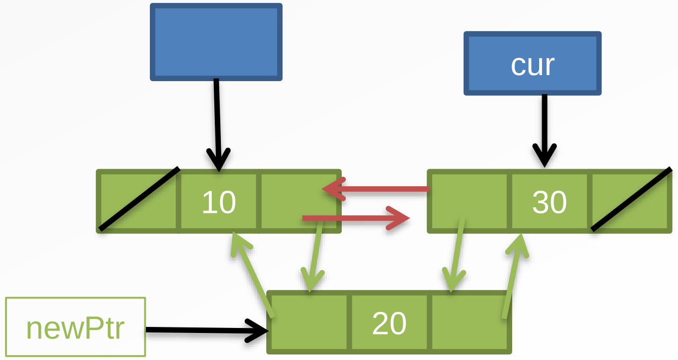
Interface Functions

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`



- Display, Search and Size functions are similar to the Singly Linked List's
- Insert function:

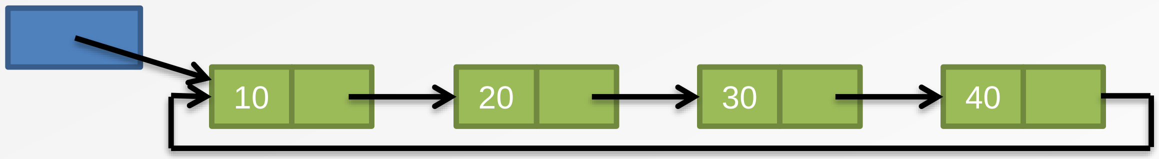
```
newPtr->next = cur;
newPtr->pre = cur->pre;
cur->pre= newPtr;
newPtr->pre->next=newPtr;
```



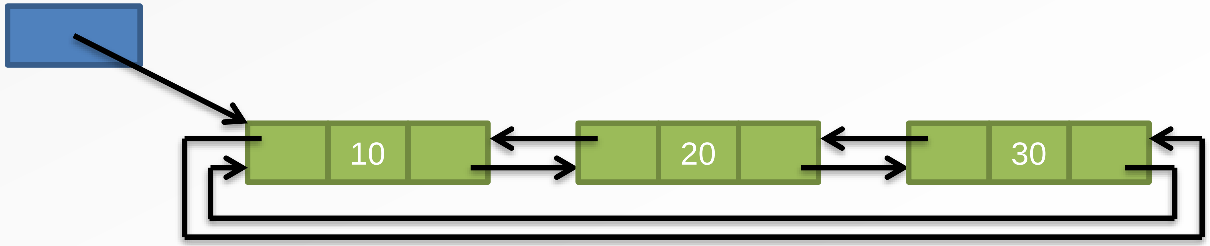
- It is noted that the solution is not unique.
- Delete function will be easier than Singly Linked List's.

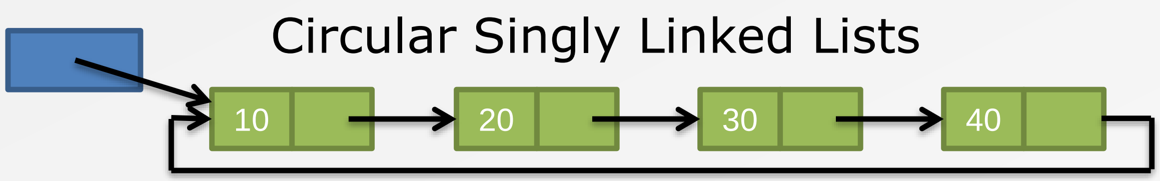
CIRCULAR LINKED LISTS

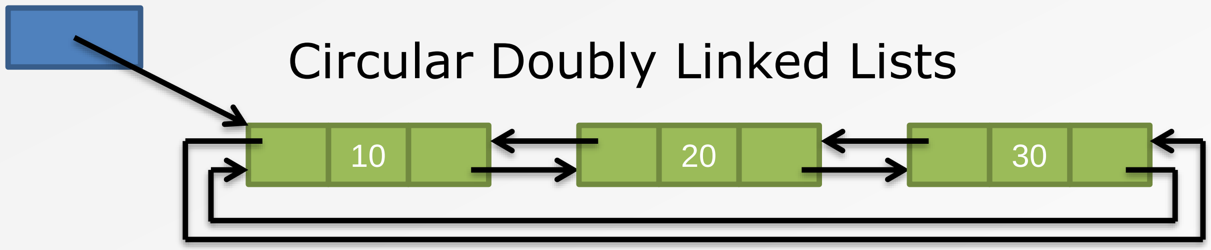
- Circular singly linked lists
  - Last node has next pointer pointing to first node



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







Interface Functions

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`

- **Display, Search, Size**: the last node's link is equal to head instead of NULL
- **Insert** and **Delete**: there is no special case at first or last position

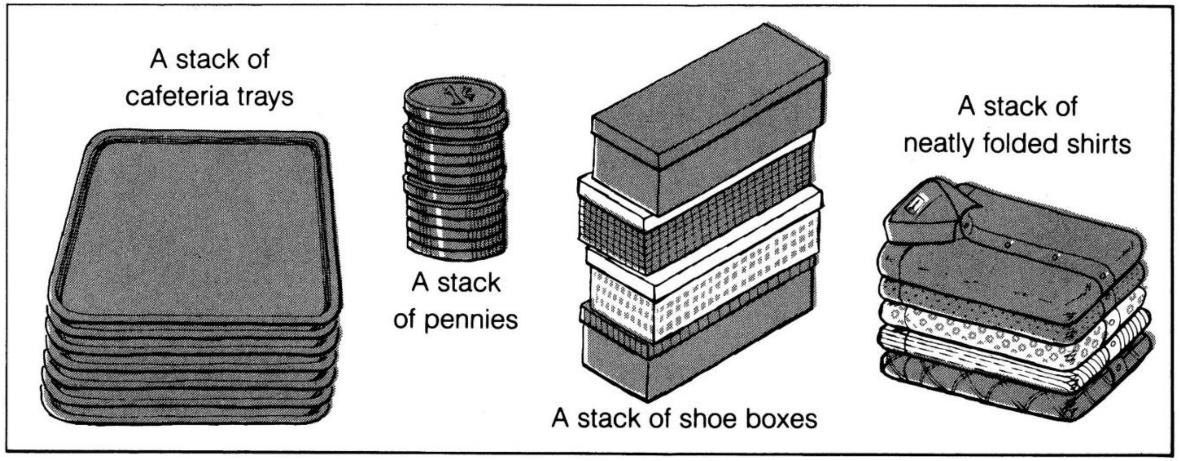
Stack and Queue

What is Stack?

What is Queue?

STACK

- Elements are added to and removed from the top



- A Last-In, First-Out (LIFO) a.k.a First-In, Last-Out (FILO) data structure
- Can be implemented by array or linked list

QUEUE

- Elements are added only at the tail and removed from the head

- A First-In, First-Out (FIFO) a.k.a Last-In, Last-Out (LILO) data structure
- Can be implemented by array or linked list

LINKED LIST TO STACK AND QUEUE

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

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(),size`

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

```
typedef ListNode StackNode;

typedef LinkedList Stack;
```

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

Stack

1. ~~Display: `printStack()`~~
2. Retrieve : `peek()`
3. Insert: `push()`
4. Delete: `pop()`
5. Size: `isEmptyStack()`

Queue

1. ~~Display: `printQueue()`~~
2. Retrieve: `getFront()`
3. Insert: `enqueue()`
4. Delete: `dequeue()`
5. Size: `isEmptyQueue()`

STACK FUNCTIONS

- Peek(): Inspect the item at the top of the stack without removing it
- Push(): Add an item to the top of the stack
- Pop(): Remove an item from the top of the stack
- IsEmptyStack(): Check if the stack has no more items remaining

- In short, users only can get access to the top of the stack. It is a FILO data structure.

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. Size: `isEmptyStack()`

peek()

- Peek the top of the stack-> return the item on the top
- Here we assume that s.head is not NULL.
- If you would like to validate s.head, then prototype of peek() need to be redefined eg. `int peek(Stack s, int* itemPtr);`

```
int peek(Stack s){
    return s.head->item;
}
```

push()

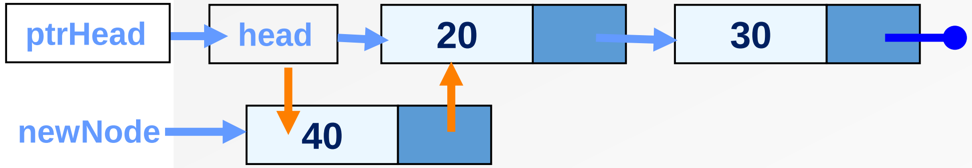
```c
int insertNode2(LinkedList *ll, int index, int item){
    ListNode  *pre, *newNode;
    if (index == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = ll->head;

        ll->head = newNode;
        ll->size++;
        return 1;
    }
    else if ((pre = findNode2(*ll, index-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        ll->size++;
        return 1;
    }
    return 0;
}
```

Stack

```c
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. **Insert: `push()`**
3. Delete: `pop()`
4. Size: `isEmptyStack()`



- Push a new node onto the stack-> insert a node at index 0

```c
void push(Stack *sPtr, int item){
    insertNode2(sPtr, 0, item);

}
```

```c
void push(Stack *sPtr, int item){
    StackNode *newNode;
    newNode= malloc(sizeof(StackNode));
    newNode->item = item;
    newNode->next = sPtr->head;
    sPtr->head = newNode;
    sPtr->size++;
}
```

pop()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`, `size`

Note:
return value of removeNode() is SUCCESS (1) or FAILURE (0)

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. **Delete: `pop()`**
4. Size: `isEmptyStack()`

```
int removeNode(LinkedList *ll, int index);
```

- Pop a node from the stack-> Remove a node at index 0 and return SUCCESS (1) or FAILURE (0)
- Here the removal node is freed directly
- Use Peek() to retrieve it first

```
int pop(Stack *sPtr){
  return removeNode(sPtr, 0);
}
```

```
int pop(Stack *s){
    if(sPtr==NULL || sPtr->head==NULL){
            return 0;
        }
        else{
            StackNode *temp = sPtr->head;
            sPtr->head = sPtr->head->next;
            free(temp);
            sPtr->size--;
            return 1;
        }
}
```

isEmptyStack()

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. **Size: `isEmptyStack()`**

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`, `size`

- Check whether the stack is empty? 1 == empty: 0 == not empty

```
int isEmptyStack(Stack s){
    if (s.size == 0) return 1;
    return 0;
}
```

LINKED LIST TO STACK AND QUEUE

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

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

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(),size`

```
typedef ListNode StackNode;

typedef LinkedList Stack;
```

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

Stack

1. ~~Display: `printStack()`~~
2. Retrieve : `peek()`
3. Insert: `push()`
4. Delete: `pop()`
5. Size: `isEmptyStack()`

Queue

1. ~~Display: `printQueue()`~~
2. Retrieve: `getFront()`
3. Insert: `enqueue()`
4. Delete: `dequeue()`
5. Size: `isEmptyQueue()`

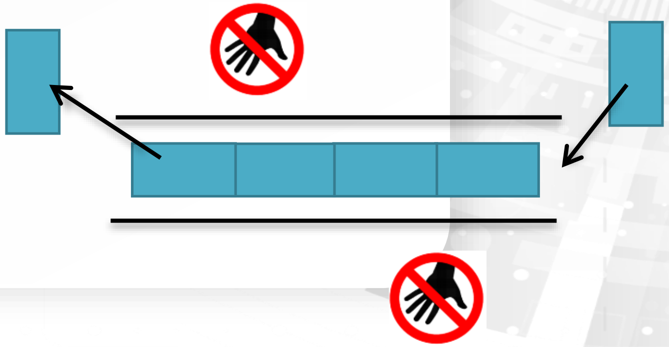
QUEUE FUNCTION

Queue

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

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

- getFront(): Inspect the item at the front of the queue without removing it
- enqueue(): Add an item at the end of the queue
- dequeue(): Remove an item from the top of the queue
- IsEmptyQueue(): Check if the queue has no more items remaining
- In short, users only can add from the back and remove from the front of the linked list. It is a FIFO data structure.
- Due to algorithmic efficiency, ***tail** is introduced



getFront()

Queue

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

1. **Retrieve: `getFront()`**
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

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

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`, `size`

- Inspect the front of the queue-> return the item at the front

```
int getFront(Queue q){
    return q.head->item;
}
```

It is same as Stack!

enqueue()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`,`size`

Queue

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

1. Retrieve: `getFront()`
2. **Insert: `enqueue()`**
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

```
int insertNode(LinkedList *ll, int index, int value);
```

- Put a new node into the Queue-> insert a node at index *size*

```
void enqueue(Queue *qPtr, int item){
 insertNode(qPtr->head, qPtr->size, item);
}
```

```
void push(Stack *s, int item){
 insertNode2(sPtr, 0, item);
}
```

- To make the `insertNode()` more efficient, *tail is introduced
- `enqueue()` needs to be rewritten to let *tail point to the last node
- Queue is empty (Size=0) is a special case

enqueue()

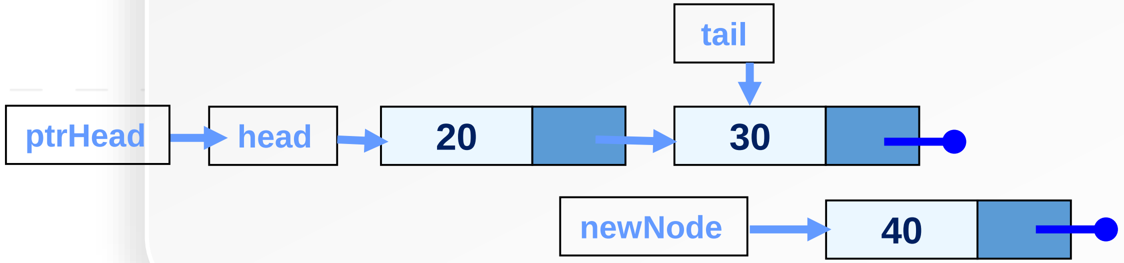
Queue

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

1. Retrieve: `getFront()`
2. **Insert: `enqueue()`**
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

- Put a new node into the Queue-> insert a node at index *size*
- To make the `insertNode()` more efficient, *tail is introduced
- `enqueue()` needs to be rewritten to let *tail point to the last node
- Queue is empty (Size=0) is a special case

```
void enqueue(Queue *qPtr, int item){
  insertNode(qPtr->head, qPtr->size, item);
}
```




```
void enqueue(Queue *qPtr, int item){
    QueueNode *newNode;
    newNode = malloc(sizeof(QueueNode));
    newNode->item = item;
    newNode->next = NULL;

    if(isEmptyQueue(*qPtr))
        qPtr->head=newNode;
    else
        qPtr->tail->next = newNode;

    qPtr->tail = newNode;
    qPtr->size++;

}
```

dequeue()

Queue

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

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. **Delete: `dequeue()`**
4. Size: `isEmptyQueue()`

- Remove a new node from the Queue-> remove a node at index 0
- *free() will not let temp ==NULL*
- **tail will point to a free memory which is not NULL*

```
int dequeue(Queue *qPtr){
    if(qPtr==NULL || qPtr->head==NULL){
            return 0;
        }
        else{
            QueueNode *temp = qPtr->head;
            qPtr->head = qPtr->head->next;
            //Queue is emptied
            if(qPtr->head == NULL)
                qPtr->tail = NULL;

            free(temp);
            qPtr->size--;
            return 1;
        }
}
```

```
int pop(Stack *sPtr){
    if(sPtr==NULL || sPtr->head==NULL){
            return 0;
        }
        else{
            StackNode *temp = sPtr->head;
            sPtr->head = sPtr->head->next;
            free(temp);
            sPtr->size--;
            return 1;
        }

}
```

It is same as Stack!

isEmptyQueue()

Queue

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

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. **Size: `isEmptyQueue()`**

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`, `size`

- Check whether the queue is empty? 1 == empty: 0 == not empty

```
int isEmptyQueue(Queue q){
    if(q.size==0) return 1;
     else return 0;
}
```

```
int isEmptyStack(Stack s){
    if (s.size == 0) return 1;
    return 0;
}
```

It is same as Stack!

STACK VS QUEUE

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

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

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. Size: `isEmptyStack()`

Queue

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

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

- push() and enqueue() are not the same
  - push() adds nodes from the head
  - enqueue() adds nodes from the tail
- Other functions are exactly doing the same things
  - dequeue() needs to take care *tail

ARRAY-BASED IMPLEMENTATION

- Stacks and queues can be implemented by linked list and array structure.
- Linked list provides more flexibility on its size
- Array allows random access
  - But you only can use head or tail in stacks and queues
- Linked list is better option