Data Structures & Algorithms

Linked Lists Variants

NUST
SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

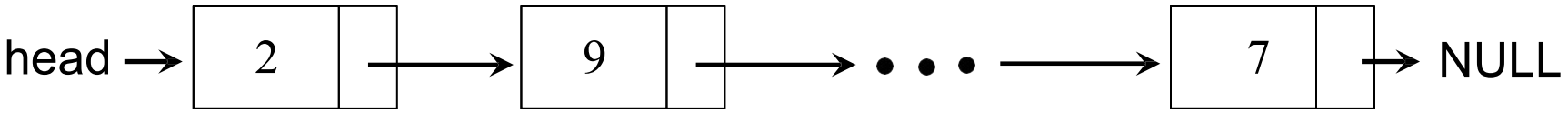
Today‘s lecture

NUST SEECS

- Linked structures
  - Singly Linked Lists
  - Doubly Linked Lists
  - Circular Linked Lists

Singly Linked List Problems

NUST SEECS



Problem:

- Cannot get backwards or to the beginning since no information related to previous node is available
  - Need a loop

```
while (nodePtr->next) // Find the last node in the list
        nodePtr = nodePtr->next;
```

Other list structures

NUST SEECS

- Doubly-linked list
  - Each node has a pointer to its successor and its predecessor
  - Faster insert/delete, but more space
- Circular list
  - The last node points back to the head
- Sorted list
  - Items stored in sorted order
- Skip list
  - Skip certain nodes to avoid sequential processing
- Self Organizing list
  - Dynamically organizes the list in a certain manner

Doubly Linked List

NUST SEECS

- Contains two references to other nodes within each node: one to the *next* node on the list, and one to the *previous* node
  - I.e., every node (except the last node) contains the address of the next node, and every node (except the first node) contains the address of the previous node
  - Allows traversal in either direction
- Implementations: ALT+TAB and ALT+SHIFT+TAB (Window Browsing)

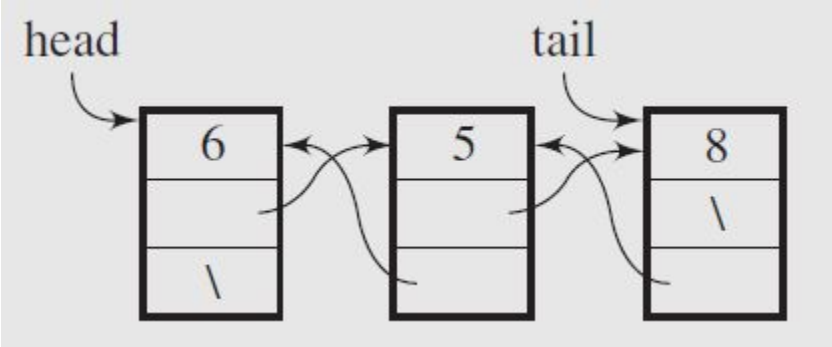
Doubly Linked List

NUST SEECS

- Add a *prev* pointer to Node class
  - Allows backward iteration

- When adding or removing a node, we must fix the prev and next pointers to have the correct value!

- Can make it easier to implement some methods such as remove



Declaration – Singly vs Doubly

NUST SEECS

Singly Linked List

```
class Node {
public:
      Node() {
            next = 0;
      }
      Node(int i, Node *in = 0) {
            info = i; next = in;
      }
      int info;
      Node *next;
};
```

Doubly Linked List

```
class Node {
public:
      Node() {
            next = prev = 0;
      }
      Node(int i, Node *n = 0 , Node *p = 0)
{
            info = i; next = n; prev = p;
      }
      int info;
      Node *next, *prev;
};
```

Add to tail operation

NUST SEECS

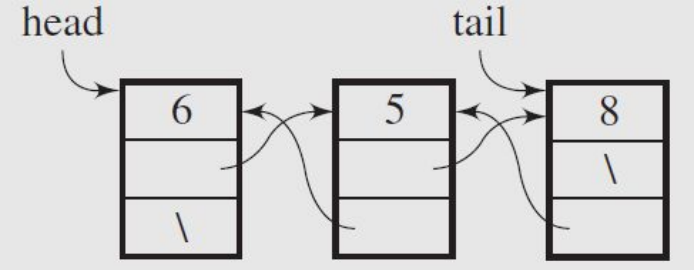
Singly Linked List

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail->next = new Node(el);
        tail = tail->next;

}
else
        head = tail = new Node(el);
}
```

Doubly Linked List

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail = new Node(el,0,tail);
        tail->prev->next = tail;

}
else
        head = tail = new Node(el);
}
```

Add to tail operation

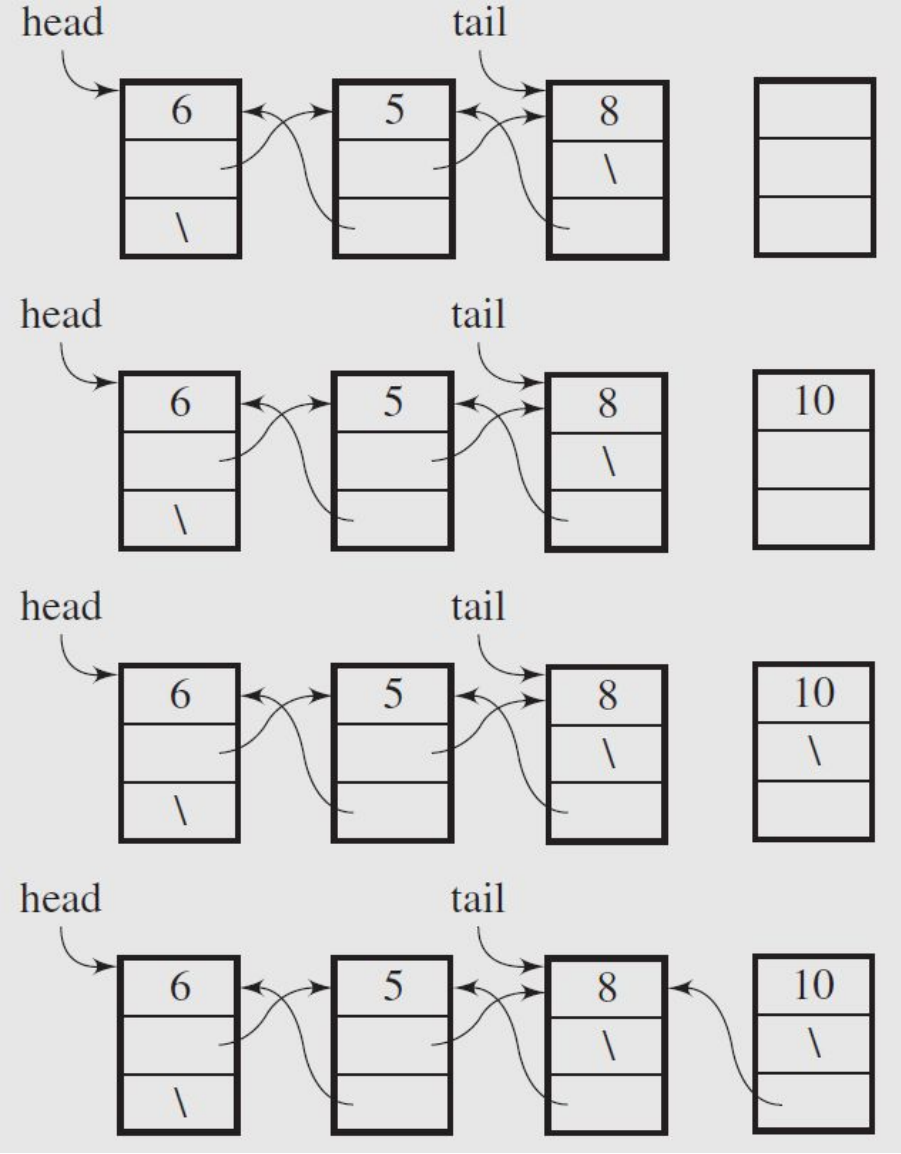


Doubly Linked List

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail = new Node(el,0,tail);
        tail->prev->next = tail;
}
else
        head = tail = new Node(el);
}
```

Call from main() with el =10

Add to tail operation



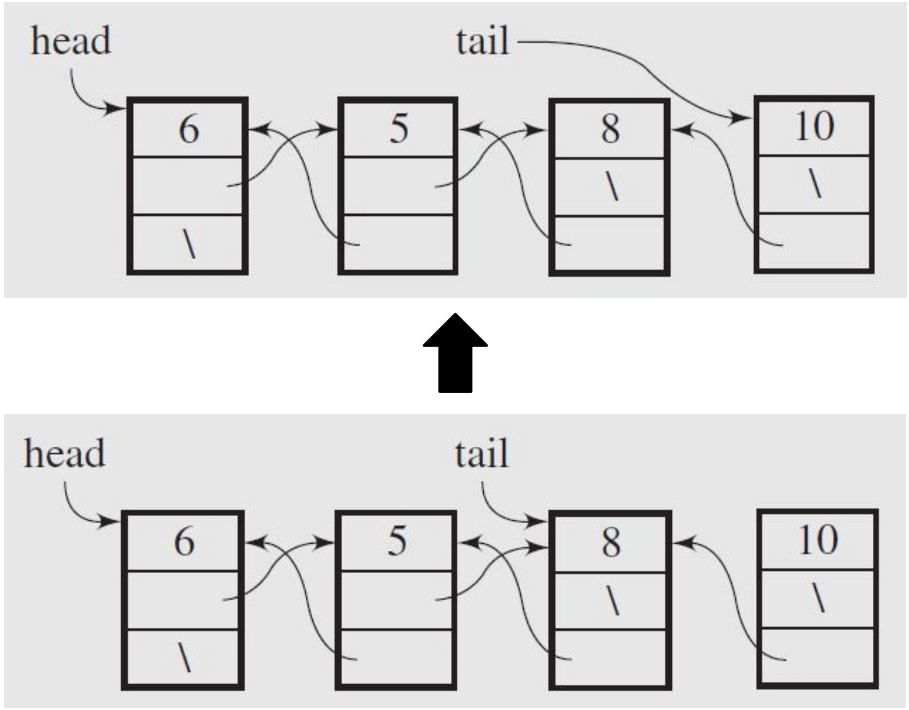
Doubly Linked List

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail = new Node(el,0,tail);
        tail->prev->next = tail;
}
else
        head = tail = new Node(el);
}
```

Call from main() with el =10

Add to tail operation

NUST SEECS

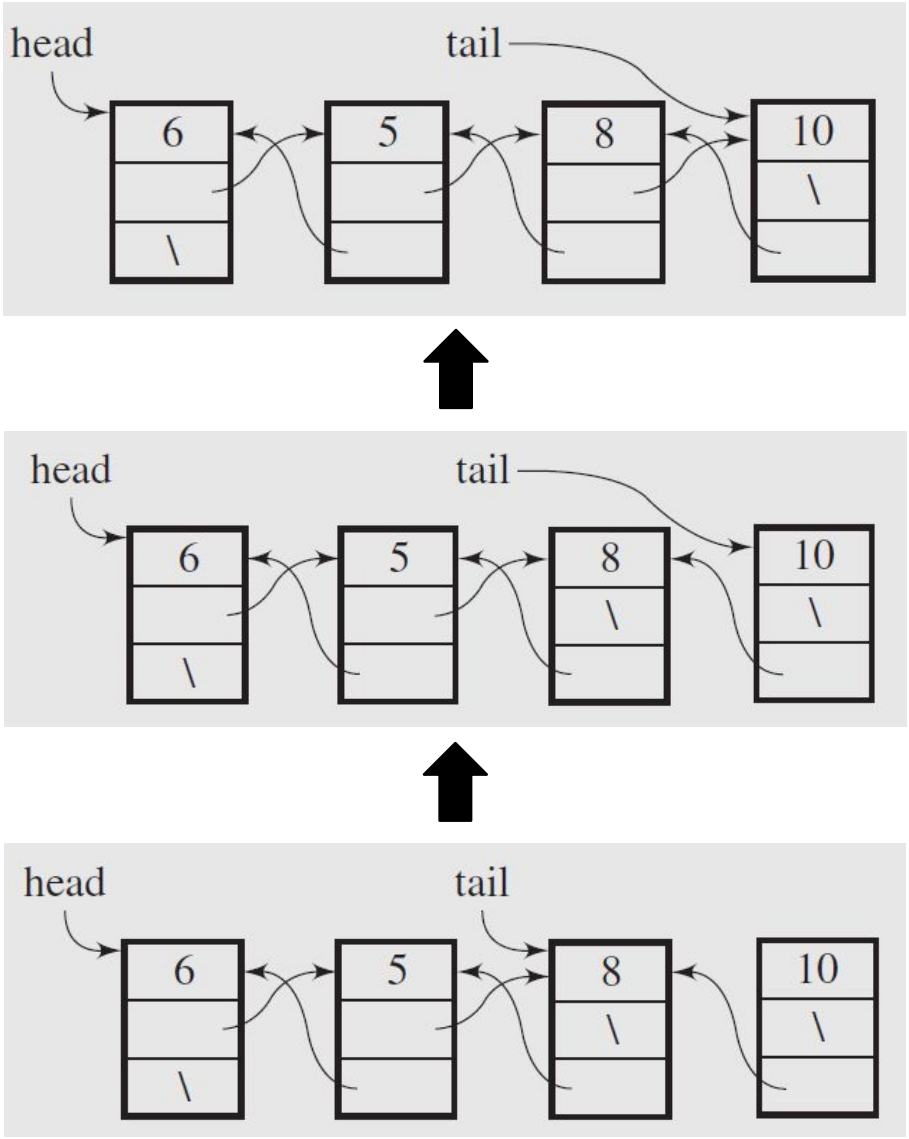


Doubly Linked List

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail = new Node(el,0,tail);
        tail->prev->next = tail;
}
else
        head = tail = new Node(el);
}
```

Call from main() with el =10

Add to tail operation



Doubly Linked List

```
void AccessNode::addToTail(int el) {
if (tail != 0) { // if list not empty;
        tail = new Node(el,0,tail);
        tail->prev->next = tail;
}
else
        head = tail = new Node(el);
}
```

Call from main() with el =10

Adding a node in middle

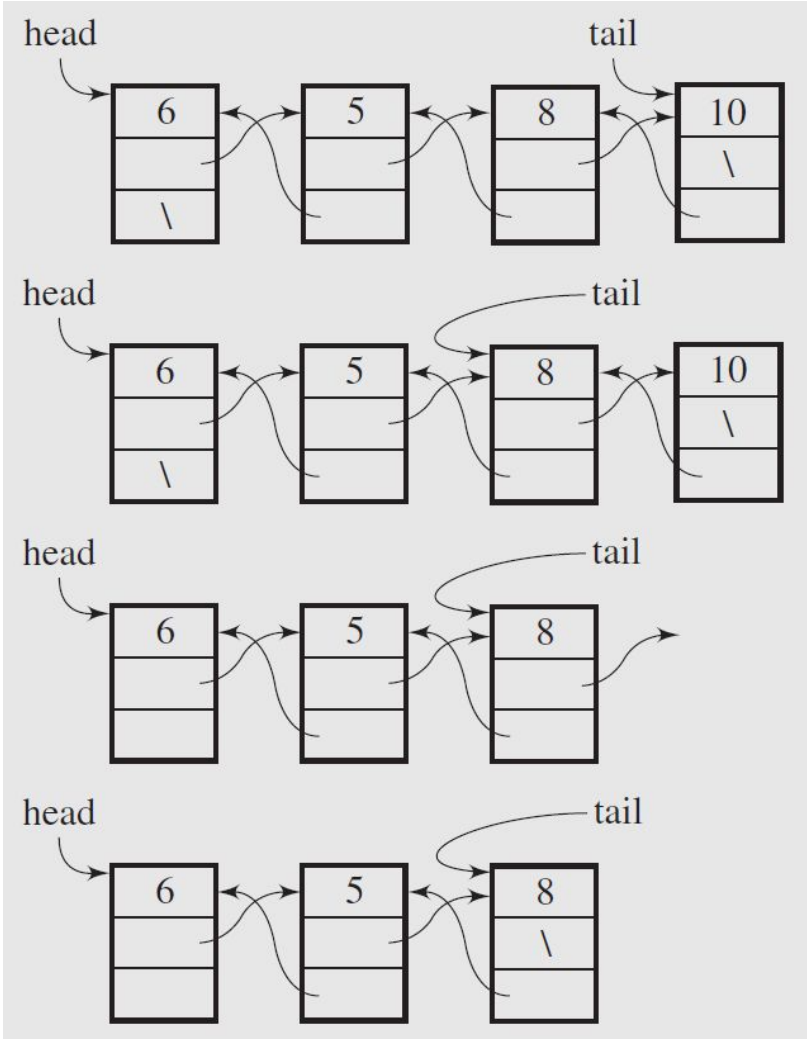
NUST SEECS

- When adding a node to the list at a given index, the following steps must be taken:
  - Advance through the list to the node just before the one with the proper index
  - Create a new node, and attach it to the nodes that should precede and follow it

Deleting a node

NUST SEECS

```
int AccessNode::deleteFromTail() {

    if (!tail) {
        // The list is empty, nothing to delete
        throw std::logic_error("Cannot delete from an
                                empty list.");
    }

    el = tail->info;

    if (head == tail) { // If only one node in the list
        delete head;
        head = tail = nullptr;
    }
    else { // If more than one node in the list
        tail = tail->prev;
        delete tail->next;
        tail->next = nullptr;
    }
    return el;
}
```



Deleting a node in middle

NUST SEECS

- When deleting a node from the list at a given index, the following steps must be taken:
  - Advance through the list to the node with the proper index
  - Detach it from the nodes that used to precede and follow it

Today‘s lecture

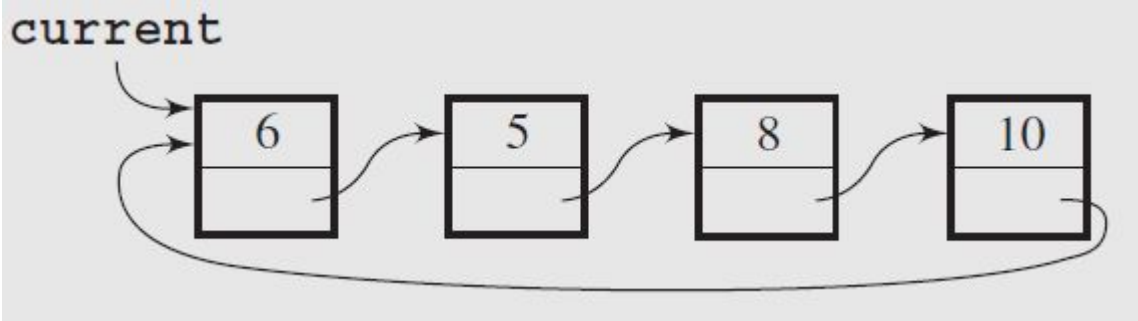
NUST SEECS

- Linked structures
  - Singly Linked Lists
  - Doubly Linked Lists
  - Circular Linked Lists

Circular Linked Lists

NUST SEECS

- Variant of Singly Linked List
  - Last node references the first node
  - Every node has a successor
  - No node in a circular linked list contains NULL

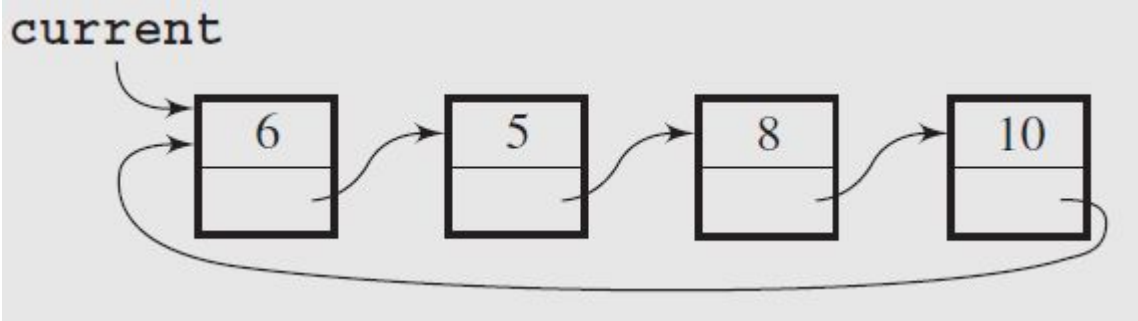


Advantage

NUST SEECS

- Allows searching from any node of the linked list and get to any other node

- Application example:
  - To ensure several processors sharing the same resource
  - **USAGE**: All processes are put on a circular list accessible through the pointer current



Implementation

NUST SEECS

- In an implementation of a circular singly linked list, we can use only one permanent pointer, i.e., **tail**, as successor of tail node is head node

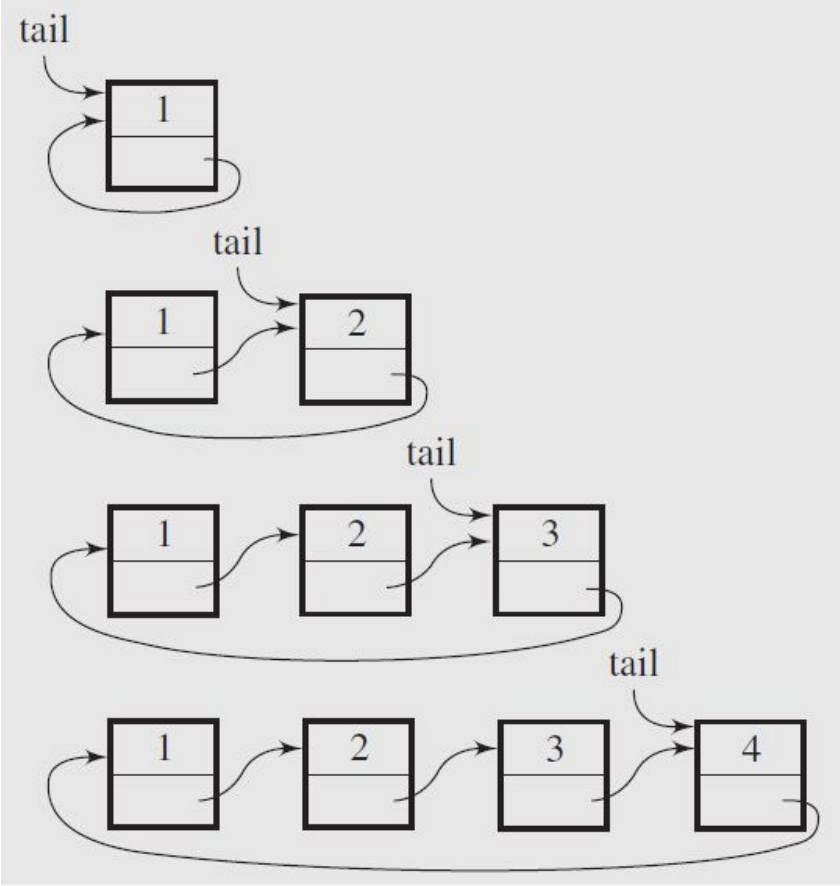
```
void addToTail(int el) {
if (isEmpty()) { // empty list
      tail = new Node(el);
      tail->next = tail;
      }
else { // not empty list
      tail->next = new Node(el,tail->next);
      tail = tail->next;
      }
}Call from main() with el =1,2,3,4
```

Implementation

NUST SEECS

- In an implementation of a circular singly linked list, we can use only one permanent pointer, i.e., **tail**, as successor of tail node is head node

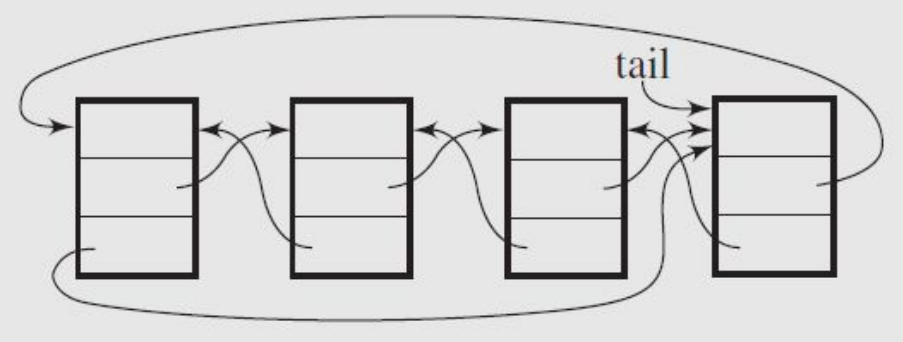
```
void addToTail(int el) {
if (isEmpty()) { // empty list
      tail = new Node(el);
      tail->next = tail;
      }
else { // not empty list
      tail->next = new Node(el,tail->next);
      tail = tail->next;
      }
}Call from main() with el =1,2,3,4
```



Problem with Circular Singly Linked List

NUST SEECS

- In deletion, still requires loop to search so that **tail** can be set to its predecessor after deleting the node
- Also, processing data in the reverse order (printing, searching, etc.) is not very efficient
- One possible solution:
  - Circular Doubly Linked List



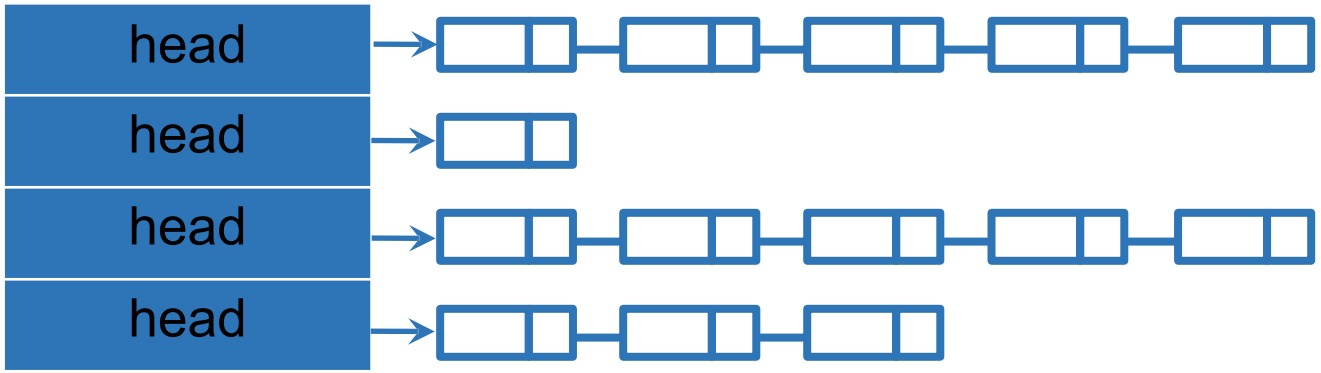
Arrays of Linked Lists

NUST SEECS

- Interesting data structure that combines static structure (an array) with dynamic structure (linked lists)

- **Example:** Array of head pointers

  - Appropriate for applications where number of categories is known in advance, but how many nodes in each category is NOT known



Arrays of Linked Lists

NUST SEECS

- Example application

  - A job portal website maintain categories and one member falls only in one category

where

  - Each category – index of array
  - Member within category – item of linked list