Revised: 12/17/2019

Linked Lists II

Michael C. Hackett

Computer Science Department

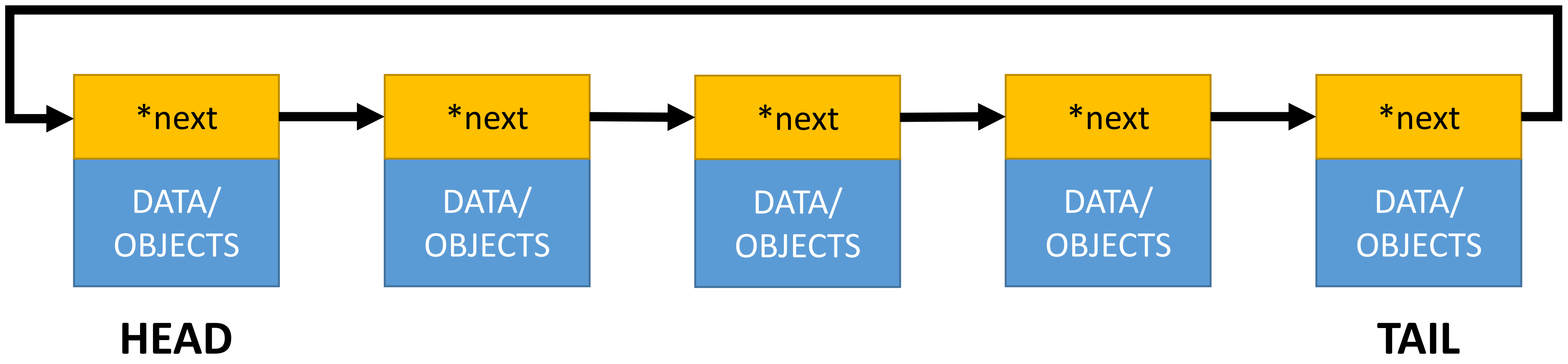
Community College of Philadelphia

Lecture Topics

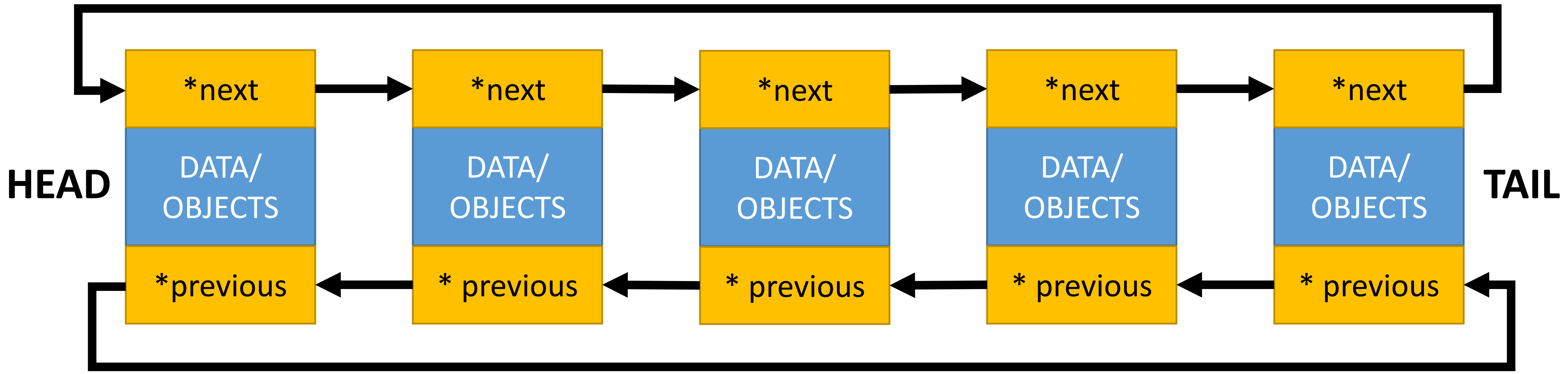
- Circular Linked Lists
  - Appending
  - Traversal
  - Prepending
  - Insertion
  - Retrieval
  - Removal
- Finding the size of a list
- Searching Linked Lists
  - Linear Search
- Sorting Linked Lists
  - Bubble Sort
  - Insertion Sort

Circular Linked List

- Each node contains a reference to the next node in the list.
- The tail node's next reference is **the head**.



Circular (Doubly) Linked List



Circular Linked Lists

- A sample struct to be used as the nodes in a list:

```
struct Node {
    int data;                    //The value stored in the node
    Node *next;                  //Pointer to the next node in the list
};
```

- In addition to the “data” field, this Node struct could contain other values, objects, or functions.
  - They can also be specified as public or private.

Circular Linked Lists

- The class itself for the Linked List will need to maintain pointers to the head and tail of the List.

```
class CLinkedList {
    private:
        Node *head;             //Pointer to the head of the list
        Node *tail;             //Pointer to the end of the list

    public:
        //Constructor. Sets the head and tail to NULL
        CLinkedList() {
            head = NULL;
            tail = NULL;
        }
}
```

Circular Linked Lists (Appending)

1. Create the new node to be added. Make sure it's next pointer is pointing to the **head** since it will be the new tail.
2. Check if head is null. If so, the list is empty; This new node is now the list's head and tail (the only node in the list)
3. Otherwise, set the current tail's next pointer to point to the new node. Set the list's tail pointer to the new node.

Circular Linked Lists (Appending)

```
void push_back(int newData) {
   Node *temp = new Node;
   temp->data = newData;
   temp->next = head;

   if(head == NULL) {
      head = temp;
      tail = temp;
   }
   else {
      tail->next = temp;
      tail = temp;
   }
}
```

Circular Linked Lists (Traversal)

- With a (singly linked) circular linked list, nodes are traversed from head to tail.

1. Start with the head node.
2. As long as it’s not null (empty list), get the node’s data.
3. Do any necessary processing with the node.
4. Use the node’s next pointer to get the next node.
5. Once we the node’s next pointer is equal to head, it means we’ve reached the end (it’s wrapping back around)

Circular Linked Lists (Traversal)

```
void printListData() {
   Node *tempPtr;
   tempPtr = head;
   do {
      cout << tempPtr->data << " ";
      tempPtr = tempPtr->next;
   } while(tempPtr != head);
   cout << endl;
}
```

Circular Linked Lists (Prepending)

1. Create the new node to be added.
2. Check if head is null. If so, the list is empty; This new node is now the list’s head and tail (the only node in the list)
3. Otherwise, make the new node’s next pointer point to the current head.
4. Set the tail node’s next pointer to point to the new node.
5. Set the list’s head pointer to the new node.

Circular Linked Lists (Prepending)

```
void push_front(int newData) {
   Node *temp = new Node;
   temp->data = newData;

   if(head == NULL) {
      head = temp;
      tail = temp;
   }
   else {
      temp->next = head;
      tail->next = temp;
      head = temp;
   }
}
```

Circular Linked Lists (Insertion)

1. Iterate to the node one place before the position where the insertion will take place
2. Check to see if it is null or is the tail (meaning we reached the end of the list/tried to go beyond the tail)
3. If it’s not, create the new node.
4. Set the new node’s next pointer to the current node’s next pointer.
5. Set the current node’s next pointer to the new node.

Circular Linked Lists (Insertion)

```
void insert(int newData, int index) {
    Node *temp = head;
    int counter = 0;
    while(counter < index-1 && temp != tail) {
        temp = temp->next;
        counter++;
    }

    if(temp == NULL || temp->next == tail) {
        return;
    }
    else {
        Node *newNode = new Node;
        newNode->data = newData;
        newNode->next = temp->next;
        temp->next = newNode;
    }
}
```

- See sample code for additional instructions that handle head and tail insertion.

Circular Linked Lists (Removal)

1. Iterate to the node one place before the position where the deletion will take place
2. Using this node, get the node two spots ahead (the one after the node to be deleted)
3. Free the node to be deleted using the free function
4. Set the previous node's next pointer to the node after the deleted node.

Circular Linked Lists (Removal)

```
void erase(int index) {
     if(index < 0 || head == NULL) {
          return;
     }
     Node *previous = head;
     if(index == 0) {
          head = previous->next;
          tail->next = head;
          free(previous);
          return;
     }
     for(int i = 0; previous != tail && i < index-1; i++) {
          previous = previous->next;
     }
     if (previous == tail) {
          return;
     }
     Node *after = previous->next->next;
     free(previous->next);
     previous->next = after;
}
```

Circular Linked Lists (Retrieval)

1. Check the list is not empty
2. Use a for loop to iterate/count to the desired node
3. Check to see if it is the head (meaning we reached the end of the list/tried to go beyond the tail)
4. Return the desired data from the node

Finding the Size of a Linked List

- Singly or doubly linked list
    1. Create a variable to keep track of the nodes visited; start at 0;
    2. Start with the head.
    3. While each node’s next pointer is not null, increment the counter
    4. Return the count.

Finding the Size of a Linked List

```
int size() {
   int count = 0;
   Node *temp = head;
   while(temp != NULL) {
      count++;
      temp = temp->next;
   }
   return count;
}
```

Finding the Size of a Linked List

- Circular linked list
    1. Create a variable to keep track of the nodes visited; start at 0;
    2. Start with the head.
    3. Check if list is empty.
    4. While each node’s next pointer is not pointing to the head, increment the counter
    5. Return the count.

Finding the Size of a Linked List

```
int size() {
   int count = 0;
   Node *temp = head;
   if(temp == NULL) {
      return count;
   }
   do {
      count++;
      temp = temp->next;
   } while(temp != head);
   return count;
}
```

Performing a Linear Search on a Linked List

- Singly or doubly linked list
  1. Start with the head.
  2. Check if it is null (list is empty)
  3. Check if current node is equal to the value being sought
  4. If it is, return it
  5. If it is not, get the next node.
  6. Repeat until the next node is null.

Performing a Linear Search on a Linked List

```
Node* linearSearch(int searchVal) {
    Node *temp = head;
    if(temp == NULL) {
        return NULL;
    }
    do {
        if(temp->data == searchVal) {
            return temp;
        }
        temp = temp->next;
    } while(temp != NULL);
    return NULL;
}
```

Circular Linked List (One Difference)

```
Node* linearSearch(int searchVal) {
    Node *temp = head;
    if(temp == NULL) {
        return NULL;
    }
    do {
        if(temp->data == searchVal) {
            return temp;
        }
        temp = temp->next;
    } while(temp != head);
    return NULL;
}
```

Performing a Bubble Sort on a Linked List

- Same algorithm can be used regardless of sorting a singly, doubly, or circular linked list.

Performing a Bubble Sort on a Linked List

```
void bubbleSort() {
    Node *temp;
    int size = this->size();
    for(int i = 0; i < size-1; i++) {
        temp = head;
        for(int j = 0; j < size-i-1; j++) {
            Node *t1 = temp;
            Node *t2 = temp->next;
            if(t1->data > t2->data) {
                int t = t2->data;
                t2->data = t1->data;
                t1->data = t;
            }
            temp = temp->next;
        }
    }
}
```

Performing an Insertion Sort on a Linked List

- Since the insertion sort goes backwards, the linked list to be sorted must be doubly linked.

Performing an Insertion Sort on a Linked List

```
void insertionSort() {
    if(head == NULL) {
        return;
    }
    int size = this->size();
    Node *sort = head->next;
    for(int i = 1; i < size; i++) {
        Node *temp = sort->previous;
        int sortValue = sort->data;
        while(temp != NULL && temp->data > sortValue) {
            int t = temp->data;
            temp->data = sortValue;
            temp->next->data = t;
            temp = temp->previous;
        }
        sort = sort->next;
    }
}
```

Circular Linked List (One Difference)

```
void insertionSort() {
    if(head == NULL) {
        return;
    }
    int size = this->size();
    Node *sort = head->next;
    for(int i = 1; i < size; i++) {
        Node *temp = sort->previous;
        int sortValue = sort->data;
        while(temp != tail && temp->data > sortValue) {
            int t = temp->data;
            temp->data = sortValue;
            temp->next->data = t;
            temp = temp->previous;
        }
        sort = sort->next;
    }
}
```