Revised: 6/9/2024

Lists I

Michael C. Hackett
Assistant Professor, Computer Science

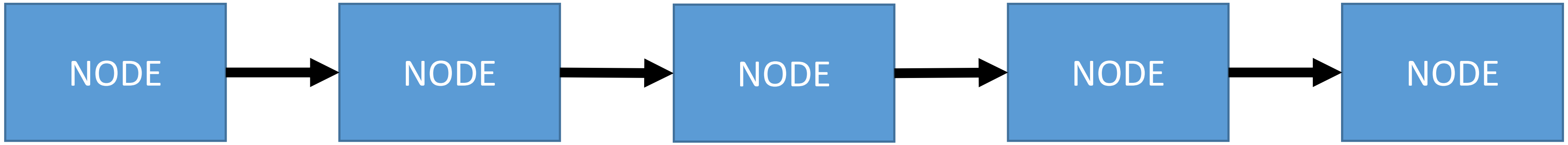
Lecture Topics

- Singly Linked Lists
- Doubly Linked Lists
- Search
- Sort

Linked Lists

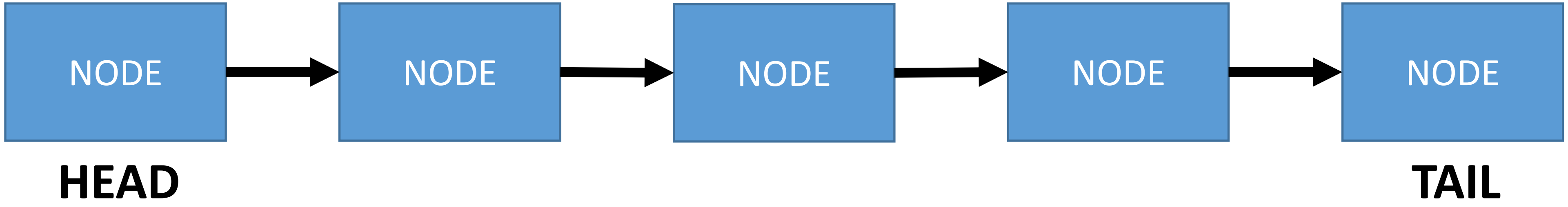
- A **linked list** is a linear data classure where a series of objects ("nodes") are connected to each other using references or other forms of references/variables.

- Each node has a reference to the next node (and in come cases the previous node) in the list.



Linked Lists

- The first node in the list is referred to as the list's **head**.
- The last node in the list is referred to as the list's **tail**.



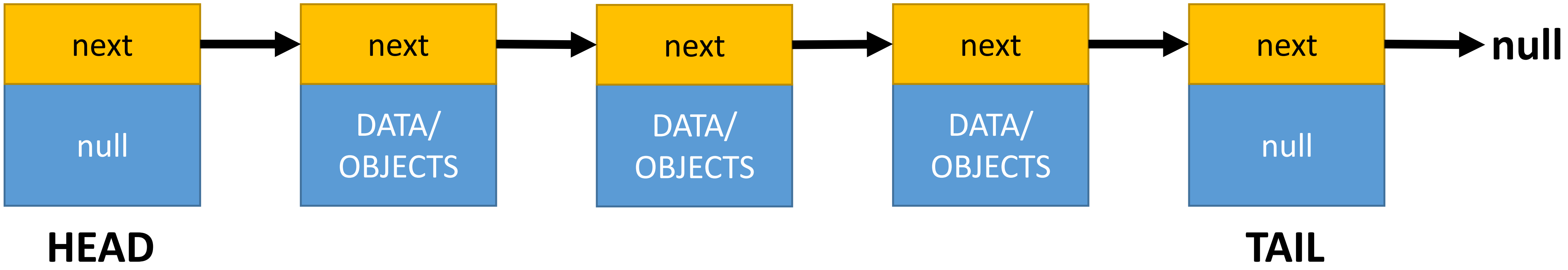
Linked Lists vs Arrays

- Arrays require the use of fixed-length, contiguous memory.

- A list does not have this requirement; Nodes can reside anywhere in memory.
  - Each node in the list knows the location of the next (and sometimes the previous) node.

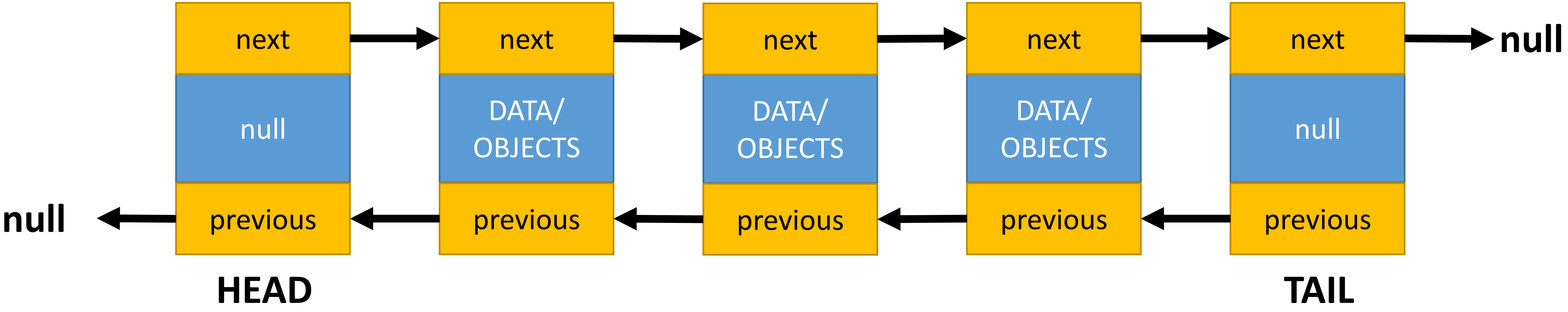
Types of Linked Lists

- Singly Linked List (SLL)
  - Each node contains a reference to the next node in the list.
  - The tail node's next reference is null.



Types of Linked Lists

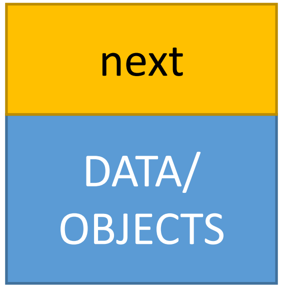
- Doubly Linked List (DLL)
  - Each node contains a reference to the next **and previous** node in the list.
  - The tail's next reference is null.
  - The head's previous reference is null.



Singly Linked Lists

- How a node for a linked list is designed depends on the application of the list.

- In the case of the singly linked list, each node will have a reference to the next node.
  - The node can contain any data or objects that it needs.



Singly Linked Lists

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

```
class Node {
    int data;                   //The value stored in the node
    Node next;                  //Reference to the next node in the list
}
```

Singly Linked Lists

- The class itself for the Linked List will need to maintain references to:
  - The head of the List
  - The tail of the List
  - The currently accessed node in the List

- The class will also keep track of the length/size of the list.

Singly Linked Lists

```
private Node head;      //Reference to the head of the list
private Node tail;      //Reference to the tail of the list
private Node current;   //Reference to the current element
private int length;     //Keeps track of the number of nodes in the list

public SLinkedList() {
    tail = new Node();          //Create tail
    current = tail;             //Current references the tail
    head = new Node();          //Create head
    head.next = tail;           //Link the two
    length = 0;
}
```

Singly Linked Lists (Appending)

- With a singly linked list, new nodes are typically added ("pushed") to the back of the list.
  1. Create the new (empty) node to be added.
  2. Set the current tail's next reference to point to the new node.
  3. Set the new data to the current tail.
  4. Update the list's tail reference to the new node.
  5. Increment the length

Singly Linked Lists (Appending)

```
public void push(int newData) {
    Node temp = new Node();        //Creates the new node to add
    tail.next = temp;              //Old tail will have a new next
    tail.data = newData;           //Put new data in old tail
    tail = tail.next;              //Set the new tail
    length++;                      //Increment the length
}
```

Singly Linked Lists (Traversal)

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

1. Start with the first node (immediately after the head).
2. If it’s not the tail (reached the end), get the node’s data.
3. Do any necessary processing with the node.
4. Use the node’s next reference to get the next node.
5. Repeat until the tail is reached.

Singly Linked Lists (Traversal)

```
public void printListData() {
   Node temp = head.next;
   while(temp != tail) {
      System.out.print(temp.data + " ");
      temp = temp.next;
   }
    System.out.println();
}
```

Singly Linked Lists (Insertion)

- While lists don't have indexes like arrays do, it's possible to insert a new node at a certain position in the list.

1. Iterate to the node at the position where the insertion will take place.
2. Create the new node.
3. Set the new node's data to the current's data (same for the "next" reference)
4. Update the current node's data and next reference to the new data and new node.
5. Check if the insertion was at the tail (update the tail reference if necessary).
6. Increment the length.

Singly Linked Lists (Insertion)

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

    if(temp == null || temp.next == null) {
        return;
    }
    else {
        Node newNode = new Node();
        newNode.data = newData;
        newNode.next = temp.next;
        temp.next = newNode;
    }
}
```

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

Singly Linked Lists (Insertion)

```
public void insert(int newData, int index) {
    moveCurrent(index);            //Move to the insertion point
    this.insert(newData);          //Insert the new data here
}

public void moveCurrent(int index) {
    //Check if valid index
    if(index < 0 || index > length) {
        throw new IndexOutOfBoundsException("Index " + index + " of out bounds.");
    }
    current = head.next;                              //Start at head
    for(int i=0; i<index; i++) {
        current = current.next;                       //Iterate to desired node
    }
}
```

Singly Linked Lists (Insertion)

```
public void insert(int newData, int index) {
    moveCurrent(index);          //Move to the insertion point
    this.insert(newData);        //Insert the new data here
}

public void insert(int newData) {
    Node temp = new Node();      //Create a new node
    temp.data = current.data;    //Place the current node's data to the new node
    temp.next = current.next;    //Make the new node point to the current node

    current.next = temp;         //Make the current node point to this node
    current.data = newData;      //Place the new data in the current node

    if(tail == current) {
        tail = current.next;     //The node inserted is the new tail
    }
    length++;
}
```

Singly Linked Lists (Removal)

- The process to remove a node is like insertion, as we need to iterate to the node immediately before the node to remove.

1. Iterate to the node at the position where the deletion will take place.
2. Put the next node's data into this node.
3. Update the tail reference, if necessary.
4. Set the current node's next reference to the node after the deleted node.
5. Decrement the length.

Singly Linked Lists (Removal)

```
public void remove(int index) {
    moveCurrent(index);                            //Move to the removal point
    this.remove();
}

public void remove() {
    if(current == tail) {
        // No current element
        throw new NoSuchElementException("No element at this index");
    }
    current.data = current.next.data;              //Pull forward the next node's data
    if(current.next == tail) {
        tail = current;                            //Removed the last element
    }
    current.next = current.next.next;              //Point around the removed node
    length--;
}
```

Singly Linked Lists (Retrieval)

- Lists aren’t indexed, and unlike arrays that use contiguous memory we can’t access elements with subscript notation like list[4]

- Lists need to iterate to the desired node/position.

Singly Linked Lists (Retrieval)

1. Move to the desired index in the list
2. Check if the current reference went to the tail.
3. Return the data from the node.

Singly Linked Lists (Retrieval)

```
public int get(int index) {
    this.moveCurrent(index);
    return this.get();
}


public int get() {
    if(current == tail) {
        // No current element
        throw new NoSuchElementException("No element at this index");
    }
    return current.data;
}
```

Doubly Linked Lists

- A sample class to be used as the nodes in a Doubly Linked List:

```
class Node {
   int data;                //The value stored in the node
   Node next;               //Reference to the next node in the list
   Node previous;           //Reference to the previous node in the list
}
```

Doubly Linked Lists

- The class itself for the Linked List will need to maintain references to:
  - The head of the List
  - The tail of the List
  - The currently accessed node in the List

- The class will also keep track of the length/size of the list.

Doubly Linked Lists

```
public class DLinkedList {

    private Node head;      //Reference to the head of the list
    private Node tail;      //Reference to the tail of the list
    private Node current;   //Reference to the current element
    private int length;     //Keeps track of the number of nodes in the list

    public DLinkedList() {
        tail = new Node();          //Create tail
        current = tail;             //Current references the tail
        head = new Node();          //Create head
        head.next = tail;           //Link the two
        tail.previous = head;       //Link the two
        length = 0;
    }

}
```

Doubly Linked Lists (Appending)

1. Create the new (empty) node to be added.
    1. Set the new data to the new node
    2. Set its previous reference to the tail’s previous node
    3. Set its next reference to the tail
2. Update the tail’s previous reference.
3. Update the next reference of the tail’s old previous node.
4. Update the current reference to now refer to the new node.
5. Increment the length

Doubly Linked Lists (Appending)

```
public void push(int newData) {
    Node temp = new Node();                   //Creates the new node to add
    temp.data = newData;                      //Sets its data
    temp.previous = tail.previous;            //Sets its previous node
    temp.next = tail;                         //Sets its next node

    tail.previous = temp;
    tail.previous.previous.next = tail.previous;
    if(current == tail) {
        current = tail.previous;
    }

    length++;                                 //Increment the length
}
```

Doubly Linked Lists (Forward Traversal)

- No different from a singly linked list.

```
public void printListForward() {
    Node temp = head.next;
    while(temp != tail) {
        System.out.print(temp.data + " ");
        temp = temp.next;
    }
    System.out.println();
}
```

Doubly Linked Lists (Backward Traversal)

- Like forward traversal, but we start with the tail and use the previous reference of each node until reaching the head

```
public void printListReverse() {
    Node temp = tail.previous;
    while(temp != head) {
        System.out.print(temp.data + " ");
        temp = temp.previous;
    }
    System.out.println();
}
```

Doubly Linked Lists (Insertion)

- The process is like insertion in a singly linked list, but we must also consider the previous reference.

1. Iterate to the node at the position where the insertion will take place.
2. Create the new node.
    1. Set the new node's data
    2. Set the new node's previous reference to the current's previous
    3. Set the new node's next reference to the current node
3. Update the current reference.
    1. Set the previous node's next reference to current (the new node)
    2. Set the next node's previous reference to current (the new node)
4. Increment the length.

Doubly Linked Lists (Insertion)

```
public void insert(int newData, int index) {
    moveCurrent(index);           //Move to the insertion point
    this.insert(newData);         //Insert the new data here
}

public void insert(int newData) {
    Node temp = new Node();                //Creates the new node to add
    temp.data = newData;                   //Sets its data
    temp.previous = current.previous;      //Sets its previous node
    temp.next = current;                   //Sets its next node

    current = temp;                        //Update current
    current.previous.next = current;       //Update previous node's next reference
    current.next.previous = current;       //Update next node's previous reference

    length++;                              //Increment the length
}
```

Doubly Linked Lists (Removal)

- Removal process is like removing of a node from a singly linked list, but we now must consider the previous references.

1. Iterate to the node at the position where the deletion will take place.
2. Update the previous node’s next reference.
3. Update the next’s node’s previous reference.
4. Update the current reference.
5. Decrement the length.

Doubly Linked Lists (Removal)

```
public void remove(int index) {
    moveCurrent(index);           //Move to the removal point
    this.remove();
}

public void remove() {
    if(current == tail) {
        // No current element
        throw new NoSuchElementException("No element at this index");
    }

    current.previous.next = current.next;       //Update previous node's next reference
    current.next.previous = current.previous;   //Update next node's previous reference
    current = current.next;                     //Update the current reference

    length--;                                   //Decrement the length
}
```

Doubly Linked Lists (Retrieval)

- Same process as the singly linked list.

```
public int get(int index) {
    this.moveCurrent(index);
    return this.get();
}


public int get() {
    if(current == tail) {
        // No current element
        throw new NoSuchElementException("No element at this index");
    }
    return current.data;
}
```

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 (or the head if circular).

Performing a Linear Search on a Linked List

```
public int linearSearch(int searchVal) {
    int index = -1;
    int counter = 0;
    Node temp = head.next;
    while(temp != tail) {
        if(temp.data == searchVal) {
            index = counter;
            break;
        }
        temp = temp.next;
        counter++;
    }
    return index;
}
```

Performing a Bubble Sort on a Linked List

- Same algorithm can be used regardless of sorting a singly or doubly linked list.
- Its easiest to swap the Nodes’ data instead of swapping the Nodes themselves.

Performing a Bubble Sort on a Linked List

```
public void bubbleSort() {
    Node temp;
    for(int i = 0; i < length-1; i++) {
        temp = head.next;
        for(int j = 0; j < length-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.

```
public void insertionSort() {
    if(length <= 1) {
        return;
    }
    Node sort = head.next.next;
    for(int i = 1; i < length; i++) {
        Node temp = sort.previous;
        int sortValue = sort.data;
        while(temp != head && temp.data > sortValue) {
            int t = temp.data;
            temp.data = sortValue;
            temp.next.data = t;
            temp = temp.previous;
        }
        sort = sort.next;
    }
}
```