Linked Lists as Stacks and Queues

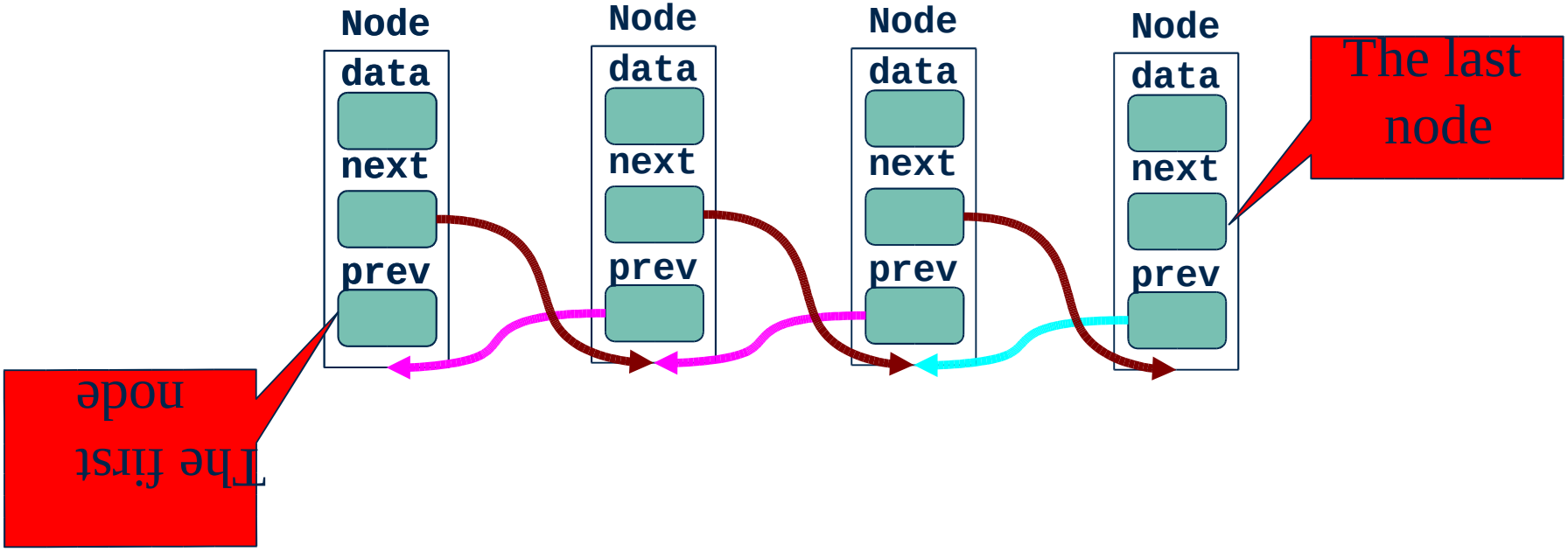
- Just as an array is used to implement stacks and queues, so can a linked list.
  - **`push`** – **`prepend`**
  - **`pop`** – **`delete(head)`**
  - **`top`** – **`head`**
  - **`enqueue`** – **`append`**
  - **`dequeue`** – **`delete(head)`**
- What is the O() of each operation when using a linked list?
  - can you do better?

Linked List Efficiency

- Insertion is very good.
- Deletion not quite so good with a linked list
- The problem is we need to traverse the list so that we can find the reference to the previous node.
  - So, not good if the node we are deleting is the tail.

Doubly-Linked Lists

- Rather than a node only having a link to its successor, it also has a link to its predecessor



Doubly Linked List: D**Node** Class

```
public class DNode {
   private int idata;           // data
   protected DNode next;
   protected DNode prev;
   public Node(int x) {
       idata = x; next = null;
   }
   public String toString() {
       return idata+"";
   }
   public void setNext(Node next) { this.next=next; }
   public void setPrev(Node next) { this.prev=next; }
   public Node getNext() { return next; }
   public Node getPrev() { return prev; }
   public int getIdata() { return idata; }
}
```

LinkedList Class

```
class DLinkedList {
  protected DNode head, tail;
  public void LinkedList() {
     head = tail = null;
  }
  public boolean isEmpty(){
     return (head == null);
  }
   public DNode first() {return head;}
   public DNode last() {return tail;}
```

`append()` method

```
public void append(Node n) {
   if (isEmpty()) {
     head = tail = n;
   }
   else {
     n.setPrev(tail);
     tail.setNext(n);
     tail = n;
   }
}
```

`prepend()` method

```
public void prepend(Node n) {
   if (isEmpty()) {
     head = tail = n;
   }
   else {
     head.setPrev(n);
     n.setNext(head);
     head = n;
   }
}
```

`toString()` method

```
public String toString() {
   if (isEmpty()) return “Empty”;
   String rtn=””;
   for (DNode tmp=tail; tmp!=null;
     tmp=tmp.getPrev())
      rtn += tmp.toString() + “ “;
   return rtn;
}
```

`delete()` method

```
public void delete(DNode n) {
 if (n==head && n==tail) {
    head=tail=null; return; }
 if (n==head) {
    head=n.getNext(); head.setPrev(null);
   return; }
 if (n==tail) {
    tail=n.getPrev(); tail.setNext(null);
    return; }
 n.getPrev().setNext(n.getNext());
 n.getNext().setPrev(n.getPrev());
 }
```

Sorted Linked List

- You can use a sorted linked list in most situations in which you would use a sorted array.
- Instead of appending or prepending a node, insert it into the list in the correct order.
- Need **`insertAfter()`** or **`insertBefore()`**

`insertAfter()` method

```
public void insertAfter(Node newNode, Node
listn) {
   newNode.setNext(listn.getNext());
   newNode.setPrev(listn);
   if (listn!=tail)
      listn.getNext().setPrev(newNode);
   else
      tail=newNode;
   listn.setNext(newNode);
}
```

`insertSorted()` method

```
public void insertSorted(DNode n) {
  DNode tmp;
  for (tmp=tail;tmp!=null;tmp=tmp.getPrev()) {
    if (tmp.getIdata() <= n.getIdata()) {
      insertAfter(n, tmp);
      return;
    }
  prepend(n);
}
```