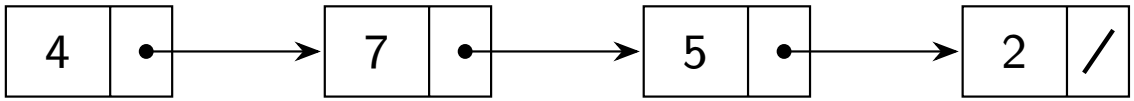
CPT108 Data Structures and Algorithms

Lecture 14

Linked Lists



Problems with Arrays

- In general, arrays are the most preferred choice to handle ordered datasets
  - It can be used to store data of different data types
  - Allow efficient creation and access to data, e.g.:
    - ★ List of 10 students marks:
      ```
      int studentMakrs[10];
      ```
    - ★ List of temperature for the last two weeks:
      ```
      double temperature[14];
      ```

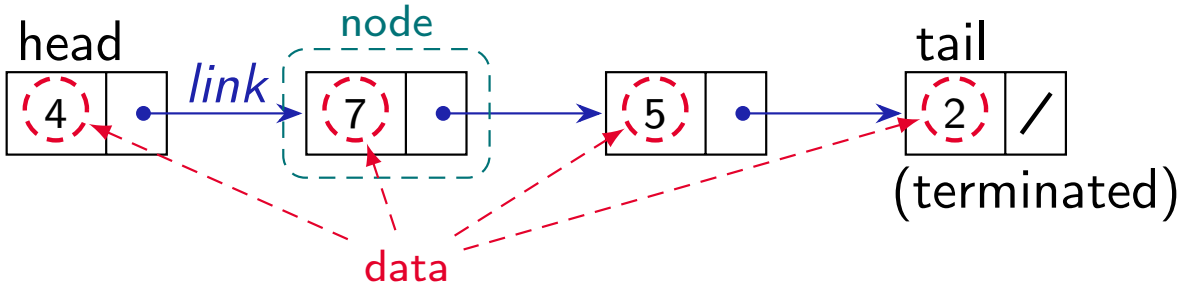
Problems with Arrays (cont.)

Disadvantages

- Fixed size – inefficient in adding new elements to, or removing elements from, the arrays
- Most programming languages require each element of a particular array to be the same size
  - In the case of large number of records, it may take more space than required for storing the same information
- In some programming languages, such as *C* and *C++*, the compiler doe not execute index bound checking, and a run time error may appear
- Have limited functionality compared to other data structures

Linked Lists: Basic Ideas

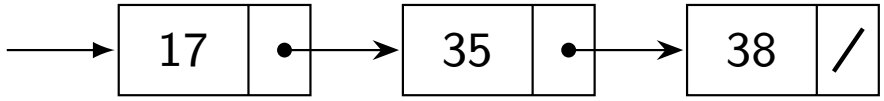
- A linked list is a series of connected elements (or nodes)
- Each element contains at least
  - A piece of **data** (of any type)
  - A **link** to the next element in the list
  - Example: A linked list of integers



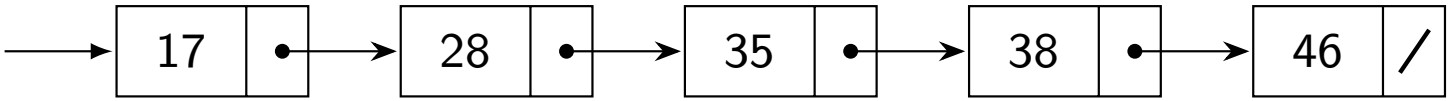
- The link is used to chain the data
- Head: pointer to the first element
- Tail: the last element of the list with its link points to `null`

Linked Lists: Basic Ideas

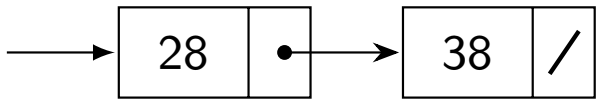
- The list can grow and shrink
- For example, in a *sorted linked list*



`add(28), add(46)`

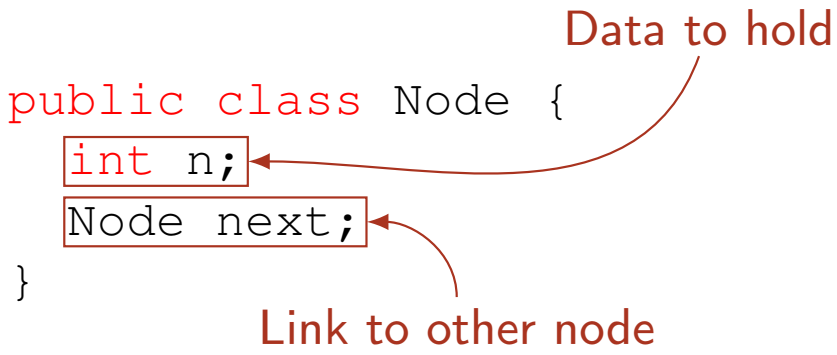


`delete(17), delete(35), delete(46)`



Linked List

A Node:



- To terminate a list:
  - Set `next` to `null`

```
Node node = new Node();
    :
node.next = null
```

- To check the termination of a list:

```
if (node.next == null) {
    ...
}
```

Linked List: Desirable list operations

- Insert a node into a list
  - In front of the list
  - In the middle of the list (after some node)
  - At the end of the list (i.e., *append*)
- Delete a node from the list
- Get the size/length of the list
- Find the leading node in the list
- Find the last node in the list
- Find whether a node is in the list
- Traverse the list
- Enumerate all nodes in the list

```
public interface List {
  int insert_head(Node node);
  int insert(Node node, int ind);
  int append(Node node);
  int set(Node node, int ind);

  int remove(Node node);

  int size();
  boolean isEmpty();

  Node first();
  Node last();

  Node get(int i);
  boolean contains(Node node);

  void clear();
}
```

Implementation

Constructor

```
public class LinkedList implements List {

  private Node head;
  private Node tail; // to append node fast
  private int size;

  public LinkedList() {
    // empty list
    head = null;
    tail = null;
    size = 0;
  }

  :
```

Implementation (cont.)

Insert and Append

```
public int insert_head(Node node) {
  node.next = head;
  head = node;
  if (tail == null)
    tail = node;
  size++;
  return size;
}

public int append(Node node) {
  if (tail != null)
    tail.next = node;
  else // empty list
    head = node;
  node.next = null;
  tail = node;
  size++;
  return size;
}
```

Implementation (cont.)

Traverse or Search a Node

- Since we only have the information of `head` and `tail`, we have to traverse, or search, a node from the head

```
public int search(Node node) {

  Node currNode = head;

  while (null != currNode && !currNode.equals(node)) {
    currNode = currNode.getNext();
  }

  if (null == currNode) return null;

  return currNode.equals(node) ? currNode : null;

}
```

Starts from the head

Traverse the list

Why needs to check this?

Linked List: Implementation (cont.)

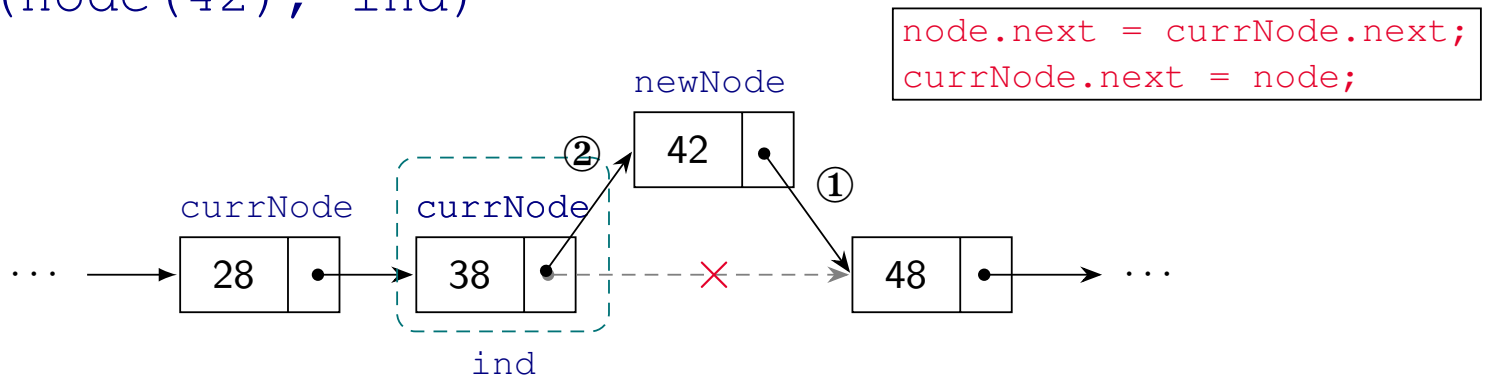
Linked list: Insert node at a specified location

```
public int insert(Node node, int ind) {
  Node currNode = head;
  int currInd = 0;

  // search for member
  while (null != currNode && ++currInd < ind) {
    currNode = currNode.getNext();
  }

  // if ind=0, insert it as the list head
  // if the requested index is bigger than the list size
  // append it to the end of the list
  if (currInd == 0) return insertHead(node);
  else if (null == currNode) return append(node);

  // find member and insert the node to list
  node.next = currNode.next;
  currNode.next = node;

  // update the tail if necessary
  if (currNode == tail) tail = node;
  size++;

  return size;
}
```

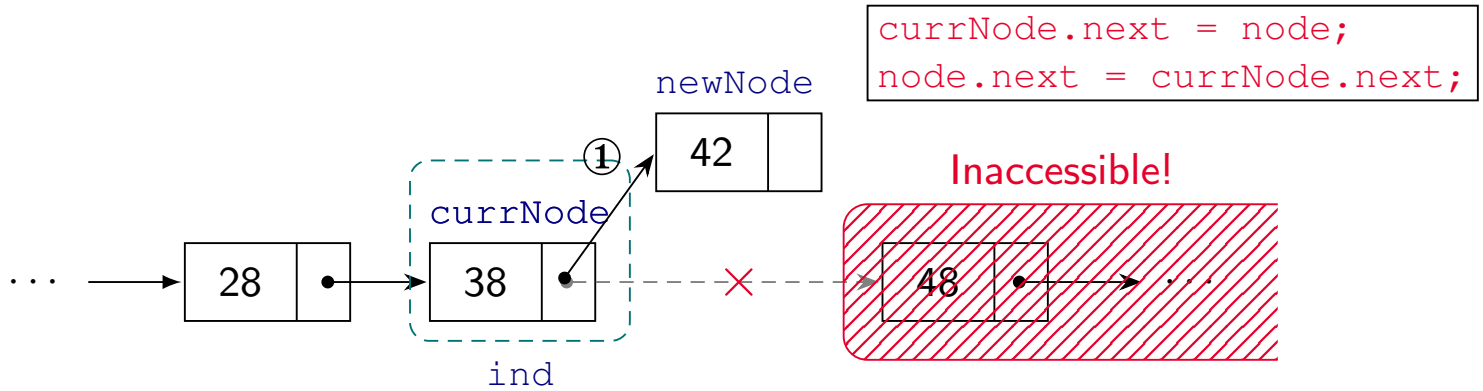
What's happening here?

Linked List: Implementation (cont.)

```
insert(node(42), ind)
```



What will happen if we swapped the order of the operations?

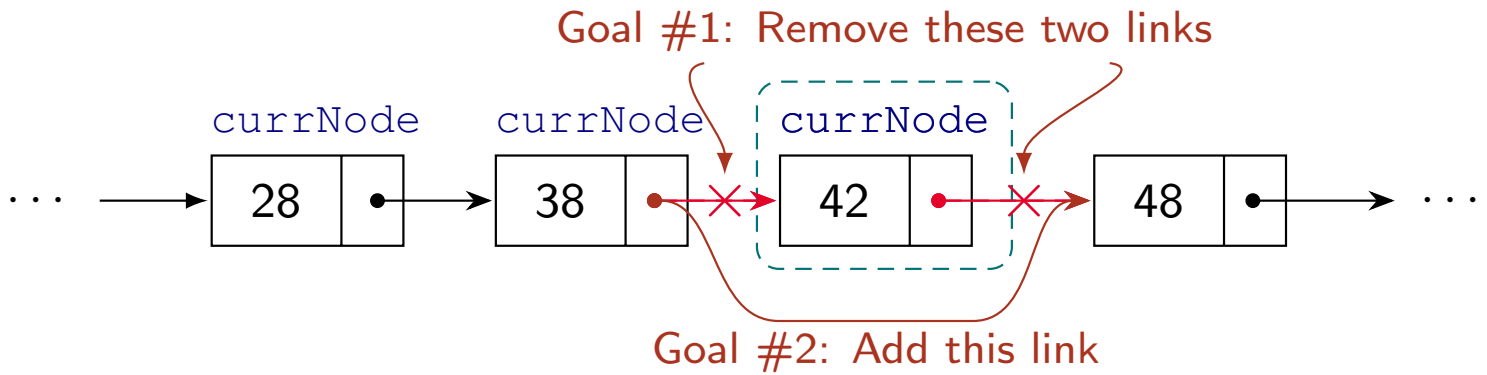


Hence, the *order of operations* in updating a linked list is very important!

Linked List: Implementation (cont.)

Linked list: Remove a node from a list

`remove(node(42))`



What should we do next?

- What link(s) should be added, updated, or removed?
  and in what order?
- Do we need other auxiliary node(s) to complete the operations?

Linked List: Implementation (cont.)

Linked list: Remove node

```
public int remove(Node node) {
  Node prevNode = null;
  Node currNode = head;

  while (null != currNode && !currNode.equals(node)) {
    prevNode = currNode;
    currNode = currNode.next;
  }

  // node not found!
  if (null == currNode) return -1;

  if (null == prevNode) { // node is the list head
    head = currNode.next;
  } else { // node is not the list head
    prevNode.next = currNode.next;
  }

  // update the tail node if necessary
  if (tail == currNode) tail = prevNode;

  // clear the node
  currNode.next = null;
  currNode = null;

  size--;
  return size;
}
```

Linked List: Exercises

Test if the given list is a palindrome:

[a b c d d c b a] - is a palindrome
[a b c d c] - not a palindrome

```
boolean isPalindrome(Node head) {
  1. create a new list in inverse order, newList
  2. check the two lists, head and newList, whether they are the
     same
}
```

Linked List: Exercises

Union of two sorted lists:

```
merge([1, 3, 4, 8], [2, 5, 7, 9])
```

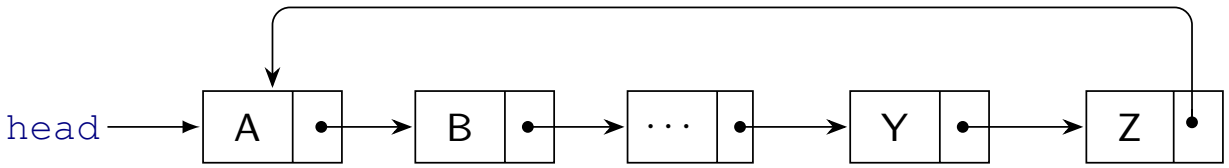
gives

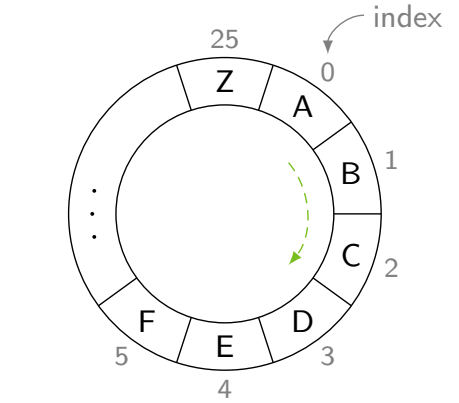
```
[1, 2, 3, 4, 5, 7, 8, 9]
```

Variants of Linked lists

Circular linked lists

- The last node points to the first node of the list, creating a *cycle*, like a *circular array*



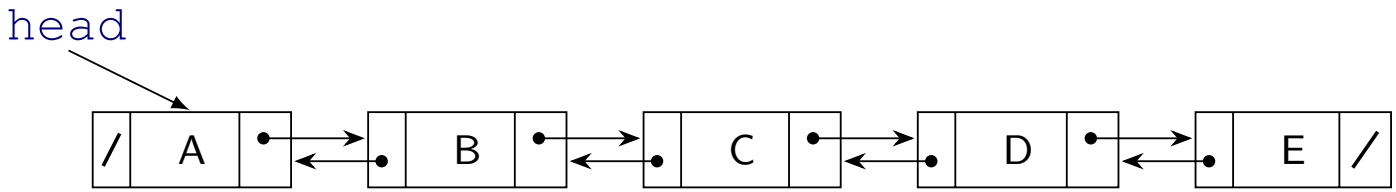


Circular representation of an array

- How do we know when we have finished traversing the list?
  - Tip: check if the pointer of the current node is equal to the head

Variants of Linked lists (cont.)

Doubly linked lists



- Each node points to not only successor, but the predecessor
- There are *two* `null` pointers: one at the first node and another one at the last node in the list
- Advantage: given a nodem it is easy to visit its predecessor. Convenient to traverse lists *backwards*

Arrays vs Linked lists

	Linked lists	Arrays
Data structure	Non-contiguous	Contiguous
Memory allocation	Dynamic	Static (fixed)
Access	Sequential	Random
Coding and Manage	Quite primitive! Need to define every operation by yourself!	Easy
Insertion/Deletion	Efficient - only need to reset some pointers, no need to move other nodes	Inefficient - may need to *resize* the array to make room for new elements or close the gap caused by deleted elements

Linked List vs Array (cont.)

time complexity

Operation	Linked lists	Arrays
Get	$O(n)$	$O(1)$
Size	$O(1)$	? (not the length!)
Set value to a particular location	$O(n)$	$O(1)$
Add(ind,value)	$O(1)$ – insert head or append $O(n)$ – otherwise	? (where is the last value? What happens if its full? and have to shifting up!)
Remove(ind)	$O(n)$	? (have to shift everything down!)
Remove(value)	$O(n)$	? (have to find the value, then shift everything down!)

Some Interesting Things...

- Java (Nov. 2023). *Choosing between ArrayList and LinkedList – JEP Cafe*. Online: https://www.youtube.com/watch?v=ul4wHrbJ8Fk
- Donald Raab (June 2023). *Sweating the small stuff in Java*. Online: https://betterprogramming.pub/sweating-the-small-stuff-in-java-dbd695166d13

Reading

- Chapter 10, Cormen (2022)

References

Thakur, Shreeya (Feb. 2024). *Advantages and Disadvantages of Array in Programming*. Online: https://unstop.com/blog/advantages-and-disadvantages-of-arrays.