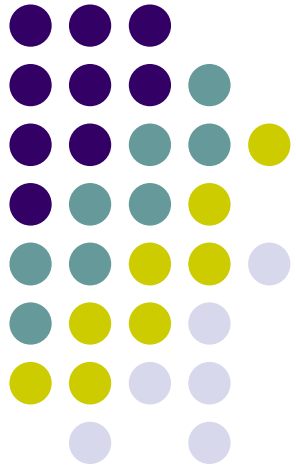
Mastering Data Structures and Algorithms: A Practical Approach

Linked Lists, Stacks and Queues

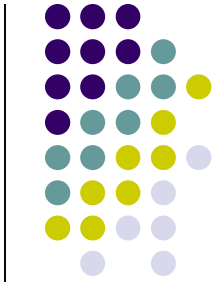


By Dr. Juan C. Gomez

Fall 2018

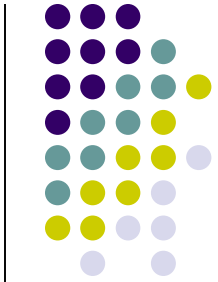
UC Santa Cruz Extension in Silicon Valley

Overview



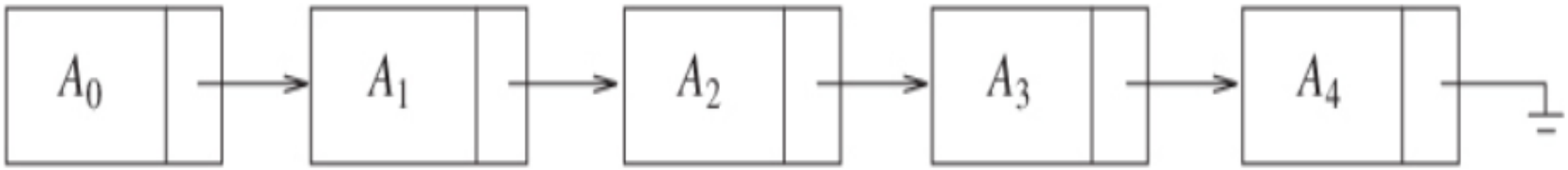
- Linked list data structure
  - Singly vs Doubly Linked list
  - Creating a Linked list
  - Deleting nodes from Singly linked list
  - The runner technique
  - Recursive problems
  - Linked lists in Java
  - Linked lists in C++
  - Examples

Overview



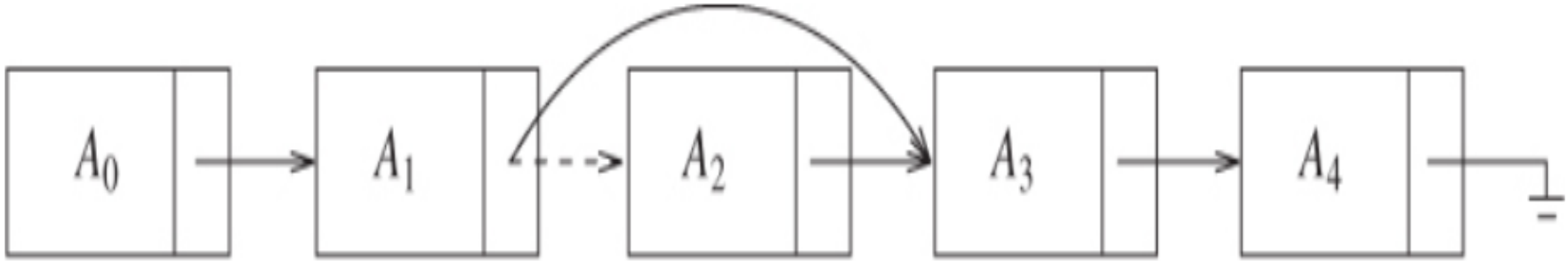
- Stacks and Queues
  - Implementing a Stack
  - Implementing a Queue
  - Stacks and Queues in C++
  - Stacks and Queues in Java
  - Examples

Singly Linked list



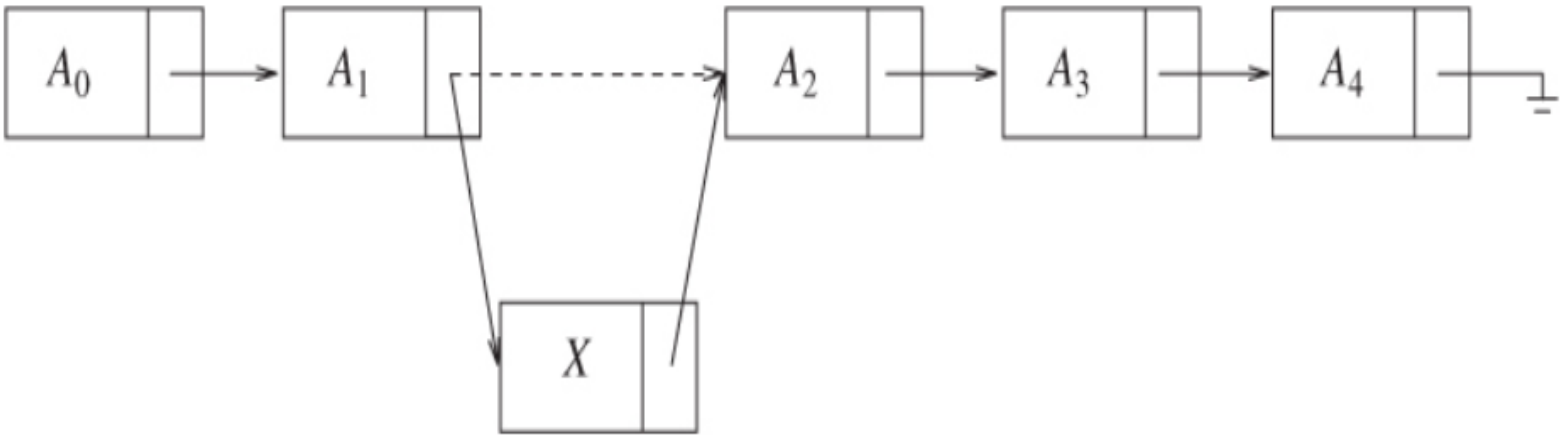
Singly Linked list

- Deletion from Singly Linked List: O(n)

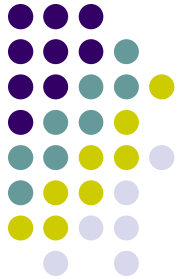


Singly Linked list

- Insertion Single Linked list: O(n)

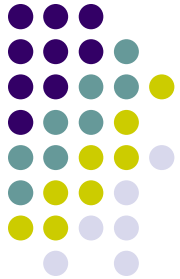


Singly Linked list



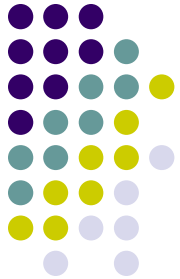
- Insertion at head Single Linked list: O(1)
- Insertion at tail Singly Linked list:
  - O(1) with pointer to tail
  - O(n) otherwise

Singly Linked list



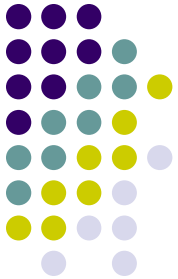
```
1   class Node {
2     Node next = null;
3     int data;
4
5     public Node(int d) {
6       data = d;
7     }
8
9     void appendToTail(int d) {
10      Node end = new Node(d);
11      Node n = this;
12      while (n.next != null) {
13        n = n.next;
14      }
15      n.next = end;
16    }
17  }
```

Singly Linked list



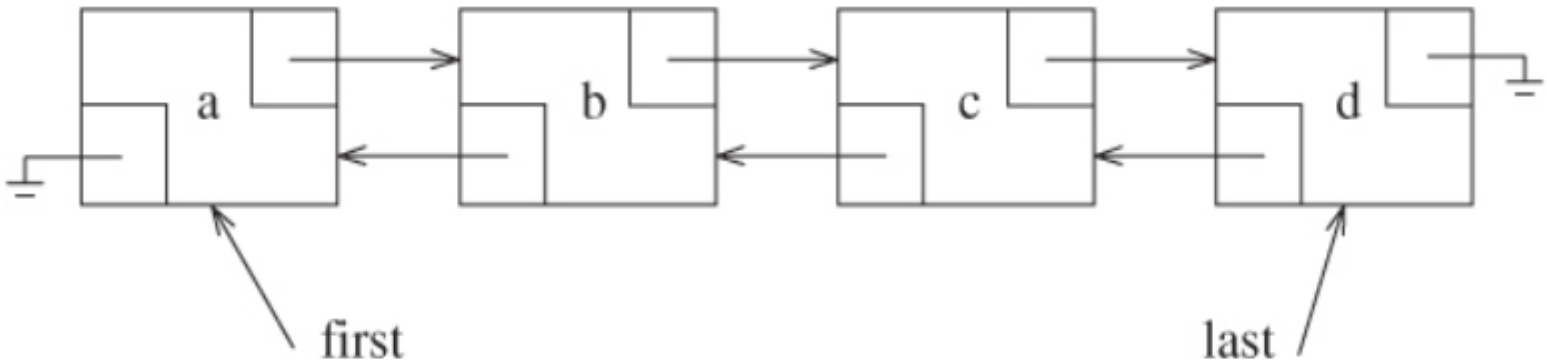
```
1   Node deleteNode(Node head, int d) {
2      Node n = head;
3
4      if (n.data == d) {
5         return head.next; /* moved head */
6      }
7
8      while (n.next != null) {
9         if (n.next.data == d) {
10           n.next = n.next.next;
11           return head; /* head didn't change */
12        }
13        n = n.next;
14     }
15     return head;
16  }
```

Singly Linked list



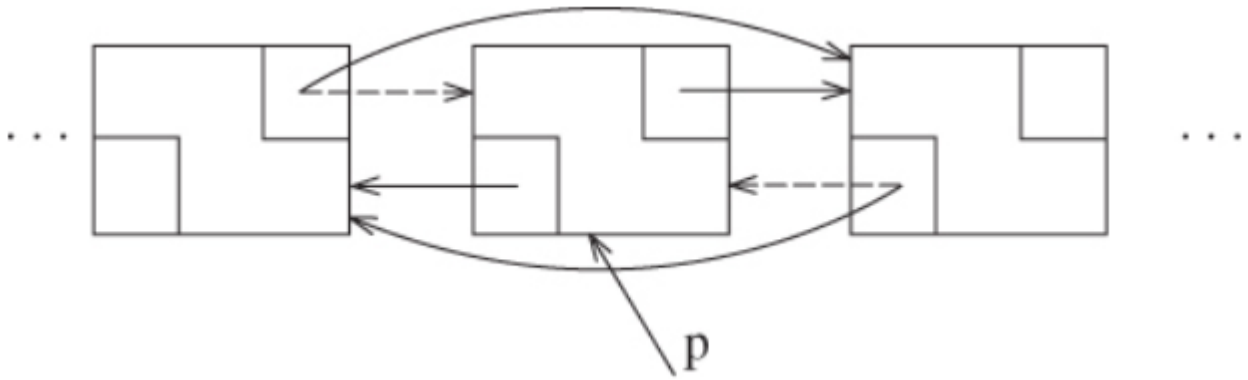
- Compare to array:
  - Dynamically allocated memory
  - Head/Tail O(1) insert time vs array O(n) head insert

Doubly Linked list



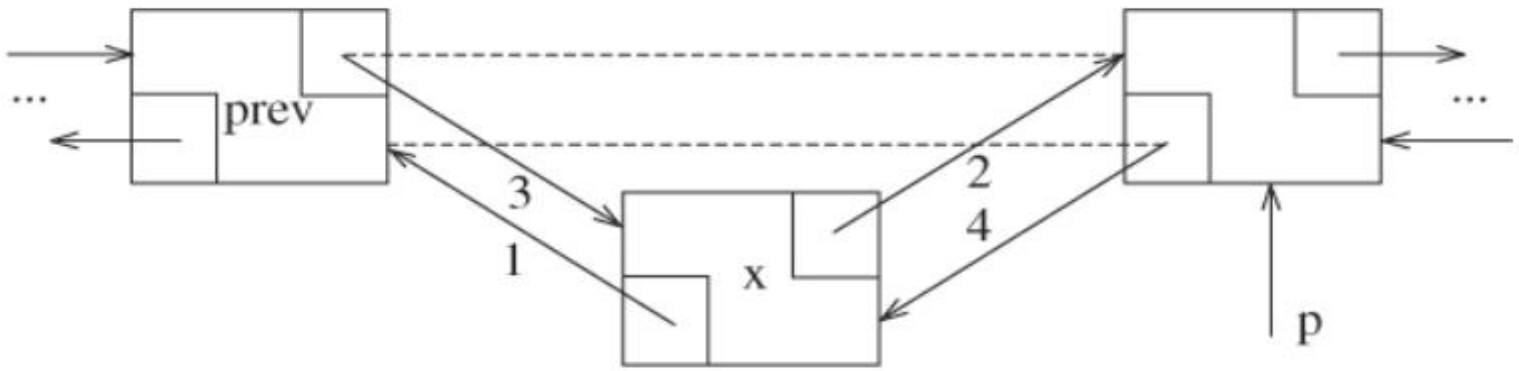
Doubly Linked list

- Doubly Linked List node delete: O(1)

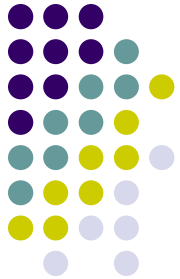


Doubly Linked list

- Doubly Linked List node insert: O(1)

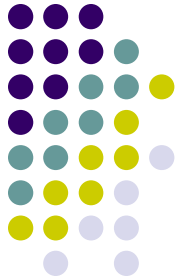


Doubly Linked list



- Compare to array, singly linked list:
  - Dynamically allocated memory
  - Head/Tail O(1) insert time vs array O(n) head insert
  - Remove with pointer to node: O(1) vs O(n) for singly linked list, array
  - Insert after with pointer to node: O(1) vs O(n) for array.
  - Find nth element: O(n) vs O(1) for an array

The “runner” technique



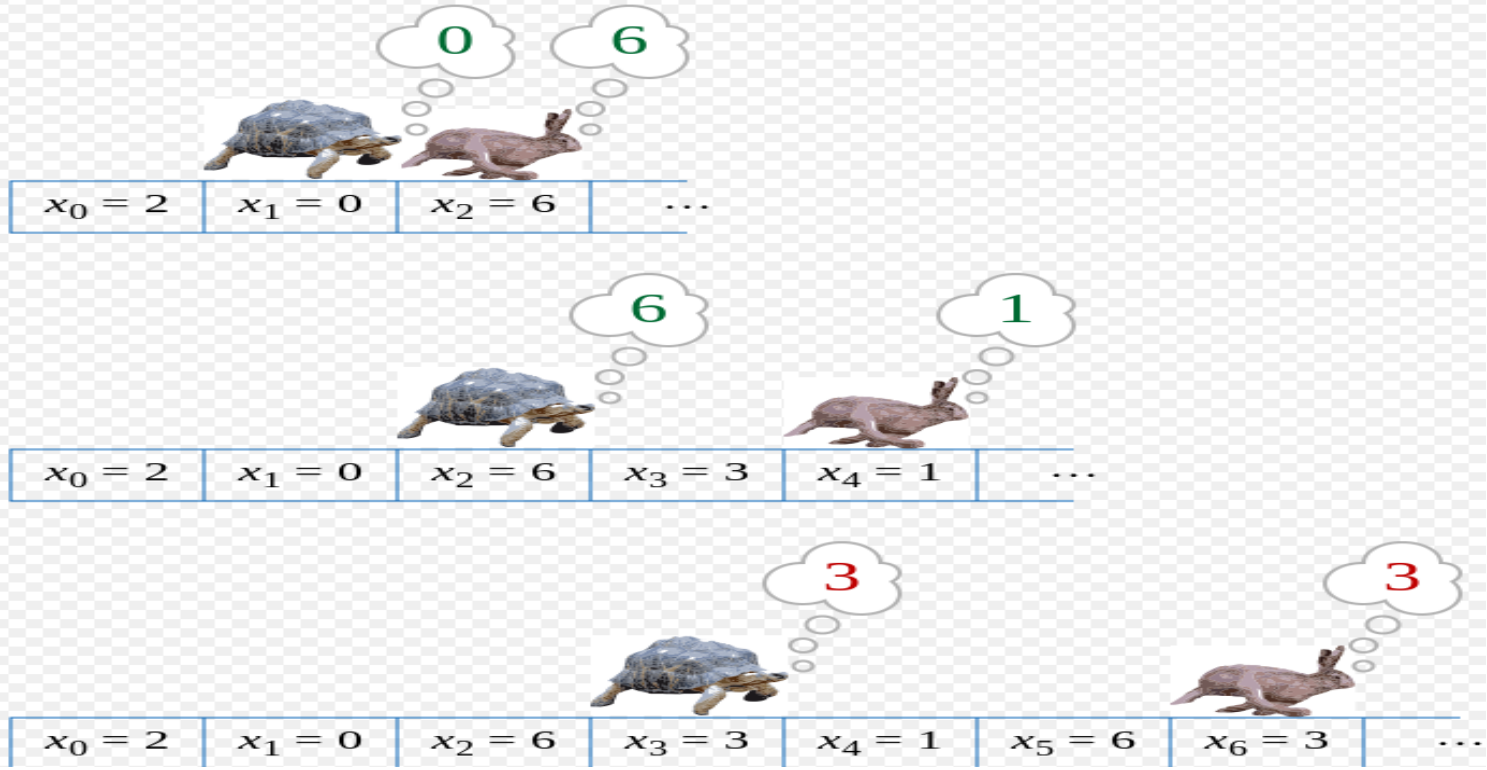
The "runner" (or second pointer) technique is used in many linked list problems. The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other. The "fast" node might be ahead by a fixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.

$a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow b_1 \rightarrow b_2 \rightarrow ... \rightarrow b_n$

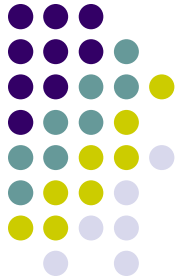
$a_1 \rightarrow b_1 \rightarrow a_2 \rightarrow b_2 \rightarrow ... \rightarrow a_n \rightarrow b_n$.

The “runner” technique

- Floyd’s Tortoise and Hare Algorithm:



Recursive List Traversal



- Pointer to current saved in the stack.
- Output Argument to return list element
- Input argument to pass previous element if needed

List in Java

- List Interface:
  - LinkedList
  - ArrayList

```
1    public interface List<AnyType> extends Collection<AnyType>
2    {
3        AnyType get( int idx );
4        AnyType set( int idx, AnyType newVal );
5        void add( int idx, AnyType x );
6        void remove( int idx );
7
8        ListIterator<AnyType> listIterator( int pos );
9    }
```

List in Java

- List Interface:
  - LinkedList => O(n)
  - ArrayList => O(n)

```
public static void makeList1( List<Integer> lst, int N )
{
    lst.clear( );
    for( int i = 0; i < N; i++ )
        lst.add( i );
}
```

List in Java

- List Interface:
  - LinkedList => O(n)
  - ArrayList => O(n^2)

```
public static void makeList2( List<Integer> lst, int N )
{
    lst.clear( );
    for( int i = 0; i < N; i++ )
        lst.add( 0, i );
}
```

List in Java

- List Interface:
  - LinkedList => $O(n^2)$
  - ArrayList => O(n)

```
public static int sum( List<Integer> lst )
{
    int total = 0;
    for( int i = 0; i < N; i++ )
        total += lst.get( i );
    return total;
}
```

List in Java

- List Interface:
  - LinkedList => $O(n^2)$
  - ArrayList => O(n)

```
public static int sum( List<Integer> lst )
{
    int total = 0;
    for( int i = 0; i < N; i++ )
        total += lst.get( i );
    return total;
}
```

List in Java

For loop

```
LinkedList<String> linkedList = new LinkedList<>();
System.out.println("==> For Loop Example.");
for (int i = 0; i < linkedList.size(); i++) {
    System.out.println(linkedList.get(i));
}
```

$O(N^2)$

Enhanced for loop

```
for (String temp : linkedList) {
    System.out.println(temp);
}
```

While loop

```
int i = 0;
while (i < linkedList.size()) {
    System.out.println(linkedList.get(i));
    i++;
}
```

$O(N^2)$

Iterator

```
Iterator Iterator = linkedList.iterator();
while (Iterator.hasNext()) {
    System.out.println(Iterator.next());
}
```

collection stream() util (Java 8)

```
linkedList.forEach((temp) -> {
    System.out.println(temp);
});
```

List in C++

Element access:

front	Access first element (public member function)
back	Access last element (public member function)

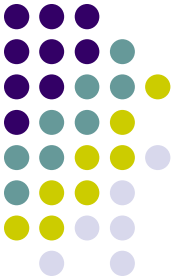
Modifiers:

assign	Assign new content to container (public member function)
emplace_front C++11	Construct and insert element at beginning (public member function)
push_front	Insert element at beginning (public member function)
pop_front	Delete first element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
emplace C++11	Construct and insert element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
resize	Change size (public member function)
clear	Clear content (public member function)

Operations:

splice	Transfer elements from list to list (public member function)
remove	Remove elements with specific value (public member function)
remove_if	Remove elements fulfilling condition (public member function template)
unique	Remove duplicate values (public member function)
merge	Merge sorted lists (public member function)
sort	Sort elements in container (public member function)
reverse	Reverse the order of elements (public member function)

Examples

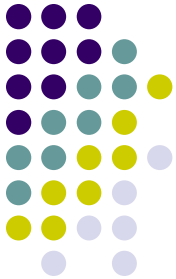


2.1 **Remove Dups:** Write code to remove duplicates from an unsorted linked list.

FOLLOW UP

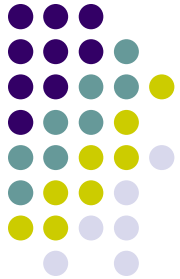
How would you solve this problem if a temporary buffer is not allowed?

Examples



2.2 Return Kth to Last: Implement an algorithm to find the kth to last element of a singly linked list.

Examples



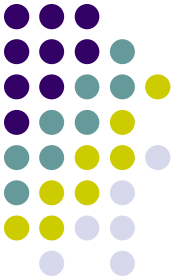
2.3 **Delete Middle Node:** Implement an algorithm to delete a node in the middle (i.e., any node but the first and last node, not necessarily the exact middle) of a singly linked list, given only access to that node.

EXAMPLE

Input: the node c from the linked list a->b->c->d->e->f

Result: nothing is returned, but the new linked list looks like a->b->d->e->f

Examples



2.4 **Partition:** Write code to partition a linked list around a value x, such that all nodes less than x come before all nodes greater than or equal to x. If x is contained within the list, the values of x only need to be after the elements less than x (see below). The partition element x can appear anywhere in the "right partition"; it does not need to appear between the left and right partitions.

EXAMPLE

Input: `3 -> 5 -> 8 -> 5 -> 10 -> 2 -> 1` [partition = 5]

Output: `3 -> 1 -> 2 -> 10 -> 5 -> 5 -> 8`

Examples

2.5 **Sum Lists:** You have two numbers represented by a linked list, where each node contains a single digit. The digits are stored in *reverse* order, such that the 1's digit is at the head of the list. Write a function that adds the two numbers and returns the sum as a linked list.

EXAMPLE

Input: (7-> 1 -> 6) + (5 -> 9 -> 2). That is, 617 + 295.

Output: 2 -> 1 -> 9. That is, 912.

FOLLOW UP

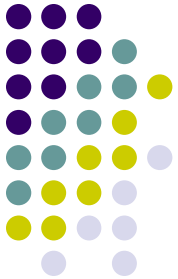
Suppose the digits are stored in forward order. Repeat the above problem.

EXAMPLE

Input: (6 -> 1 -> 7) + (2 -> 9 -> 5). That is, 617 + 295.

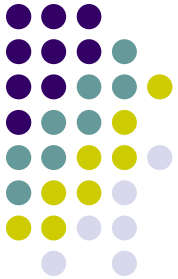
Output: 9 -> 1 -> 2. That is, 912.

Examples



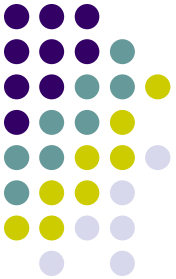
2.6 Palindrome: Implement a function to check if a linked list is a palindrome.

Examples



2.7 **Intersection**: Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the kth node of the first linked list is the exact same node (by reference) as the jth node of the second linked list, then they are intersecting.

Examples



2.8 **Loop Detection:** Given a circular linked list, implement an algorithm that returns the node at the beginning of the loop.

DEFINITION

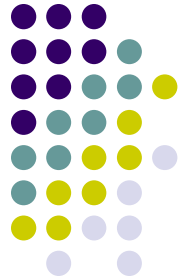
Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: A -> B -> C -> D -> E -> C [the same C as earlier]

Output: C

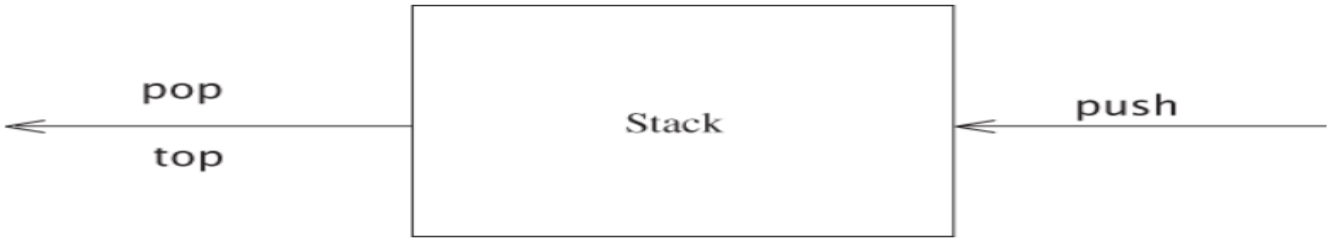
Classical Linked List Problems



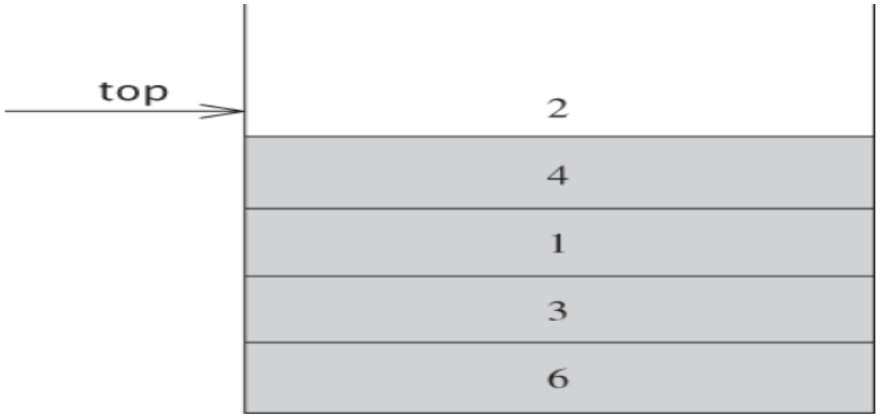
- http://cslibrary.stanford.edu/105/LinkedListProblems.pdf

Stacks and Queues

- Stack: Last-In First-Out (LIFO) data structure

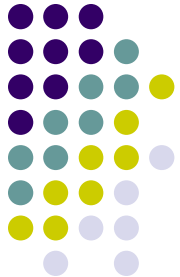


Stack model: Input to a stack is by `push`; output is by `pop` and `top`



Stack model: Only the top element is accessible

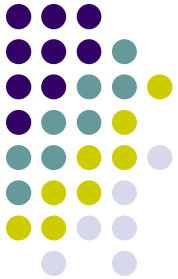
Stacks and Queues



- Stack: Last-In First-Out (LIFO) data structure

  - `pop()`: Remove the top item from the stack.
  - `push(item)`: Add an item to the top of the stack.
  - `peek()`: Return the top of the stack.
  - `isEmpty()`: Return true if and only if the stack is empty.

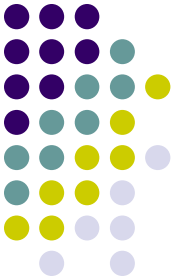
Stacks and Queues



- Simple Stack Implementation:

```
1   public class MyStack<T> {
2      private static class StackNode<T> {
3         private T data;
4         private StackNode<T> next;
5
6         public StackNode(T data) {
7            this.data = data;
8         }
9      }
10
11     private StackNode<T> top;
12
13     public T pop() {
14        if (top == null) throw new EmptyStackException();
15        T item = top.data;
16        top = top.next;
17        return item;
18     }
19
```

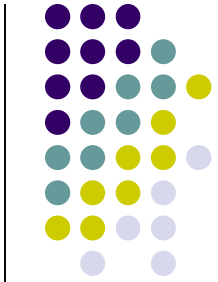
Stacks and Queues



- Simple Stack Implementation:

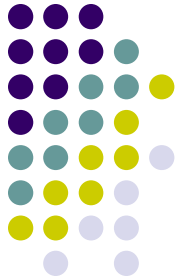
```
20    public void push(T item) {
21       StackNode<T> t = new StackNode<T>(item);
22       t.next = top;
23       top = t;
24    }
25
26    public T peek() {
27       if (top == null) throw new EmptyStackException();
28       return top.data;
29    }
30
31    public boolean isEmpty() {
32       return top == null;
33    }
34 }
```

Stacks and Queues



- Simple Stack Implementation:
  - Can be implemented using a Linked List
  - Can be implemented using an array

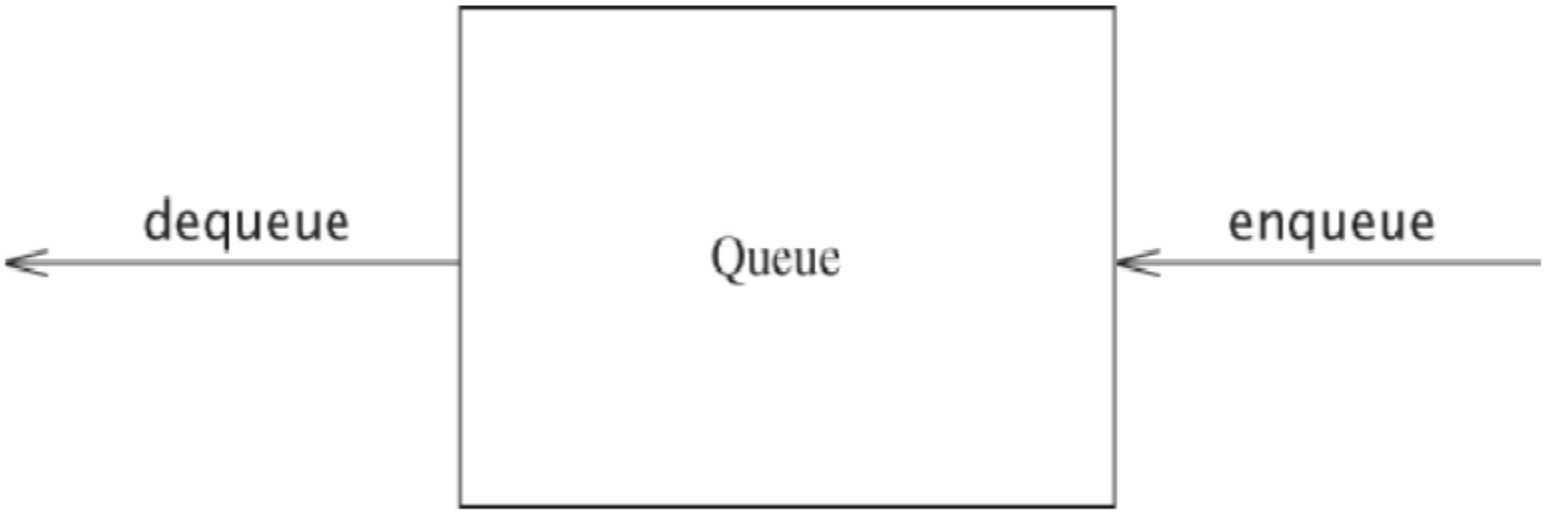
Stacks and Queues



- Stack Uses:
  - Balancing Symbols
  - Postfix Expressions
  - Infix to Postfix conversion
  - Recursive to Iterative
  - Some tree traversals
  - Save state as you recurse, reuse state when unrolling recursion.

Stacks and Queues

- Queue: First-In First-Out (FIFO) data structure



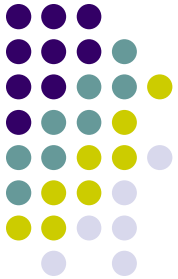
Model of a queue

Stacks and Queues

- Queue: First-In First-Out (FIFO) data structure

  - `add(item)`: Add an item to the end of the list.
  - `remove()`: Remove the first item in the list.
  - `peek()`: Return the top of the queue.
  - `isEmpty()`: Return true if and only if the queue is empty.

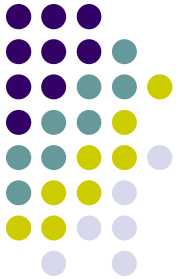
Stacks and Queues



- Simple Queue Implementation:

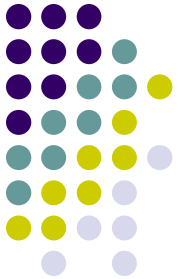
```
1   public class MyQueue<T> {
2     private static class QueueNode<T> {
3       private T data;
4       private QueueNode<T> next;
5
6       public QueueNode(T data) {
7         this.data = data;
8       }
9     }
10
11    private QueueNode<T> first;
12    private QueueNode<T> last;
13
```

Stacks and Queues



```
14      public void add(T item) {
15         QueueNode<T> t = new QueueNode<T>(item);
16         if (last != null) {
17            last.next = t;
18         }
19         last = t;
20         if (first == null) {
21            first = last;
22         }
23      }
24
25      public T remove() {
26         if (first == null) throw new NoSuchElementException();
27         T data = first.data;
28         first = first.next;
29         if (first == null) {
30            last = null;
31         }
32         return data;
33      }
34
35      public T peek() {
36         if (first == null) throw new NoSuchElementException();
37         return first.data;
38      }
39
40      public boolean isEmpty() {
41         return first == null;
42      }
43   }
```

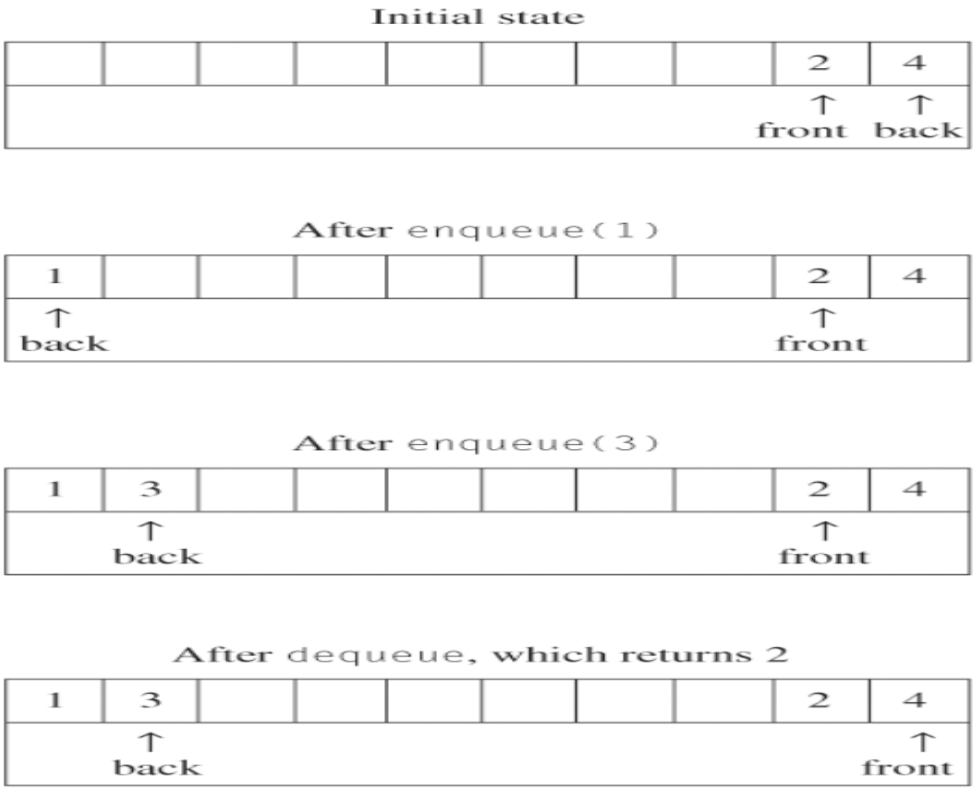
Stacks and Queues



- Simple Queue Implementation:
  - Can be implemented using a Linked List
  - Can be implemented using an array

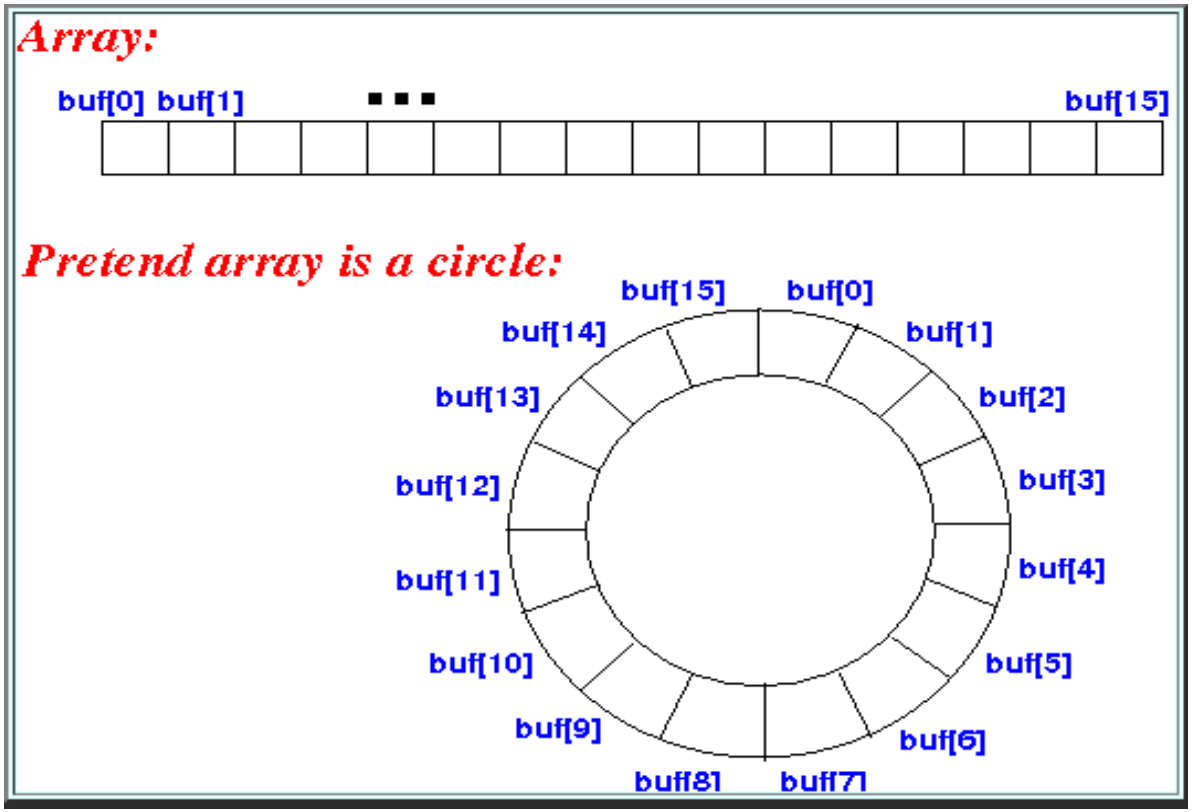
Stacks and Queues

- Array Implementation:



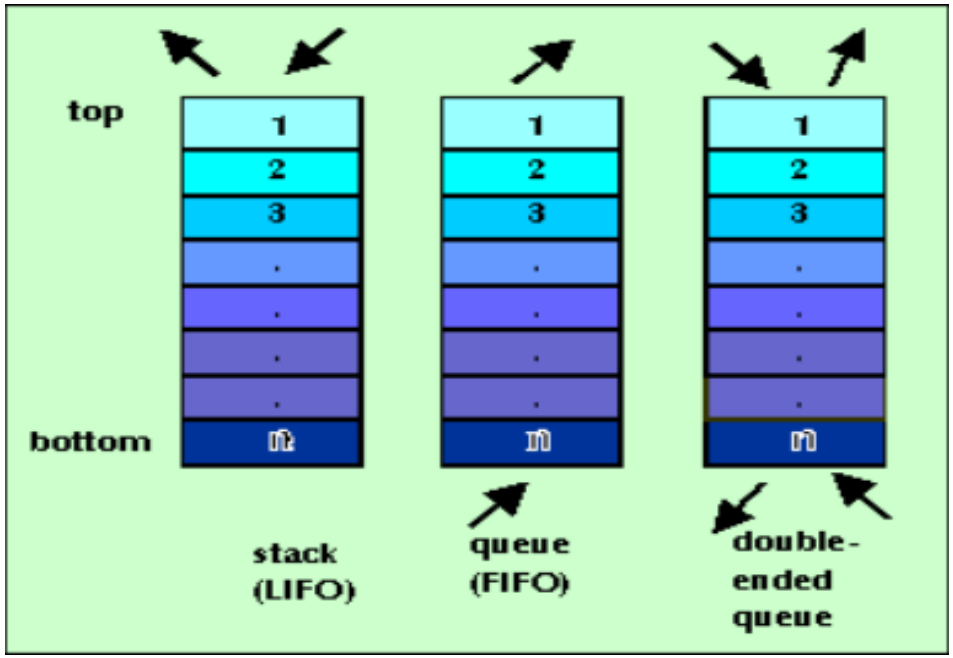
Stacks and Queues

- Interesting Data Structure:

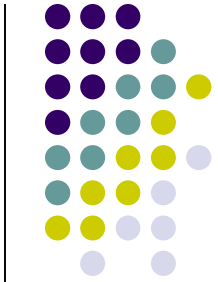


Stacks and Queues

- Dequeues: double ended queue
  - Elements can be added/removed from the head or tail.

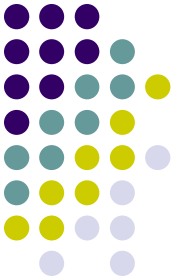


Stacks and Queues



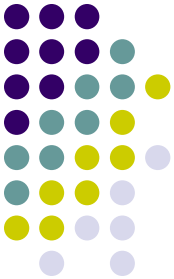
- Uses of Queues:
  - Breadth-first search
  - Job scheduling
  - Networking

Stacks and Queues in C++



```
1  // stack::push/pop
2  #include <iostream>       // std::cout
3  #include <stack>          // std::stack
4
5  int main ()
6  {
7    std::stack<int> mystack;
8
9    for (int i=0; i<5; ++i) mystack.push(i);
10
11   std::cout << "Popping out elements...";
12   while (!mystack.empty())
13   {
14      std::cout << ' ' << mystack.top();
15      mystack.pop();
16   }
17   std::cout << '\n';
18
19   return 0;
20 }
```

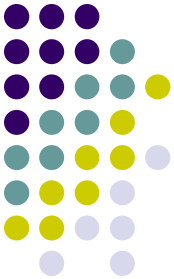
Stacks and Queues in C++



fx Member functions

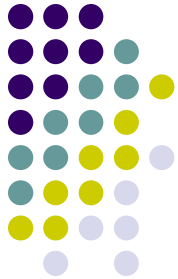
(constructor)	Construct stack (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access next element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap C++11	Swap contents (public member function)

Stacks and Queues in C++



```
1  // queue::push/pop
2  #include <iostream>       // std::cin, std::cout
3  #include <queue>          // std::queue
4
5  int main ()
6  {
7    std::queue<int> myqueue;
8    int myint;
9
10   std::cout << "Please enter some integers (enter 0 to end):\n";
11
12   do {
13     std::cin >> myint;
14     myqueue.push (myint);
15   } while (myint);
16
17   std::cout << "myqueue contains: ";
18   while (!myqueue.empty())
19   {
20     std::cout << ' ' << myqueue.front();
21     myqueue.pop();
22   }
23   std::cout << '\n';
24
25   return 0;
26 }
```

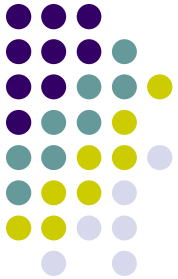
Stacks and Queues in C++



fx Member functions

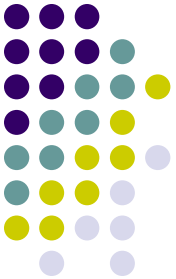
(constructor)	Construct queue (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
front	Access next element (public member function)
back	Access last element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove next element (public member function)
swap C++11	Swap contents (public member function)

Examples



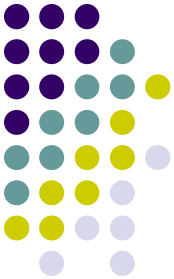
3.1 Three in One: Describe how you could use a single array to implement three stacks.

Examples



3.2 Stack Min: How would you design a stack which, in addition to push and pop, has a function `min` which returns the minimum element? Push, pop and `min` should all operate in O(1) time.

Examples

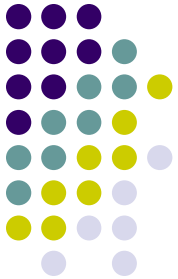


3.3 **Stack of Plates:** Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

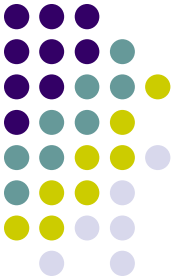
Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

Examples



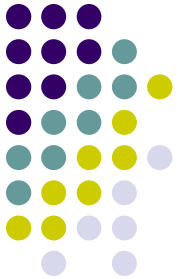
3.4 Queue via Stacks: Implement a MyQueue class which implements a queue using two stacks.

Examples



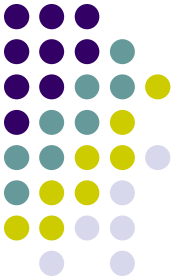
3.5 **Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

Examples



3.6 **Animal Shelter:** An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog, and dequeueCat. You may use the built-in `LinkedList` data structure.

Classical Queue/Stacks Problems



- Tower of Hanoi
- Postfix notation Evaluator
- Check symbol balancing