Common Data Structures

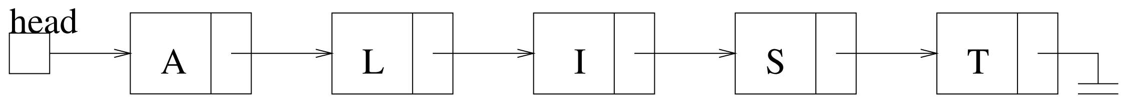
- Arrays (single and multiple dimensional)
- **Linked Lists**
- **Stacks**
- **Queues**
- Trees
- Graphs

You should already be familiar with *arrays*, so they will not be discussed. *Trees* and *Graphs* will be discussed later.

Linked Lists

A **linked list** is a linear set of *nodes* with the following properties:

- Each node has at least two fields, the *key* and the *next*.
- The next field of the ith element "points to" the $(i+1)$th element of the list.
- The first element of the linked list is called the *head*, and the last element the *tail*. The head contains no data, and the next field of the tail points to *NULL*.
- The *key* field contains the data we are actually interested in.
- What exactly is meant by things like "points to" and "NULL" depends on the implementation used.



Linked List Operations

There are several operations that we *must* be able to do on a linked list:

- Insert a node
- Delete a node
- Find an element with key k

There are other operations which might be useful:

- Move an element
- Swap two elements

We will talk about a few of these next, and give one possible way to implement a linked list (in part, anyway).

Linked List: C++ Declaration

```
struct node
  { char key;
    struct node *next; }
struct node *head;
head = new node;
head->next = NULL;
```

- This creates an empty linked list.
- In this implementation, we have the tail point to NULL, which is usually defined to be ‘0’. Thus, if the next field of an element is ‘0’, we know we have reached the tail.
- Here “point to” means that. We really used pointers. We will see shortly that we don’t need to use actual pointers to implement a linked list.

Linked List: JAVA Declaration

- The JAVA implementation is similar to the C++ one.

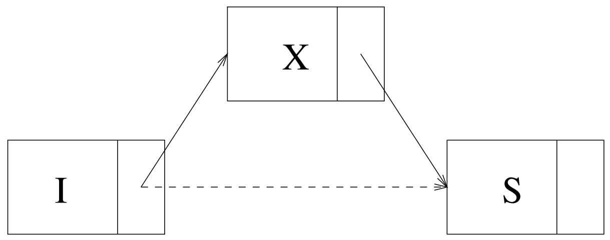
```
public class node {
    public char key;
    public node next;
    }
node head = new node();
head.next = NULL;
```

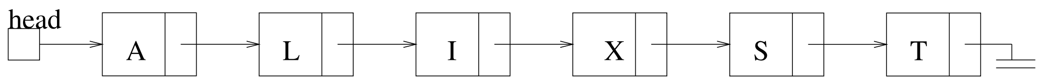
Linked List: Insert

- The simplest insert would always place the new item at the front (or back) of the list. One may also want to insert elements in the middle of the list.
- To insert X between I and S:

```
//C++
struct node *A;
A=new node;
A->key = X;
A->next = I->next;
I->next = A;
```

```
//JAVA
node A;
A=new node();
A.Key=X;
A.next=I.next;
I.next=A;
```



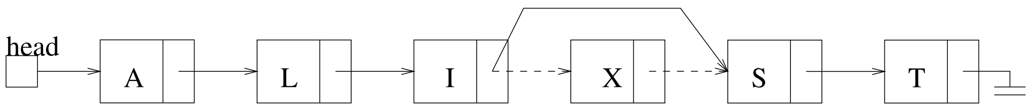


- Only two references need to be changed no matter how long the list is. How does this compare with arrays?

Linked List: Delete

- Deleting a node is very simple. We just need to change one pointer.
- However, we need to know which node points to the node we wish to delete.
- Assuming we know that node i points to node x, we can delete x by:

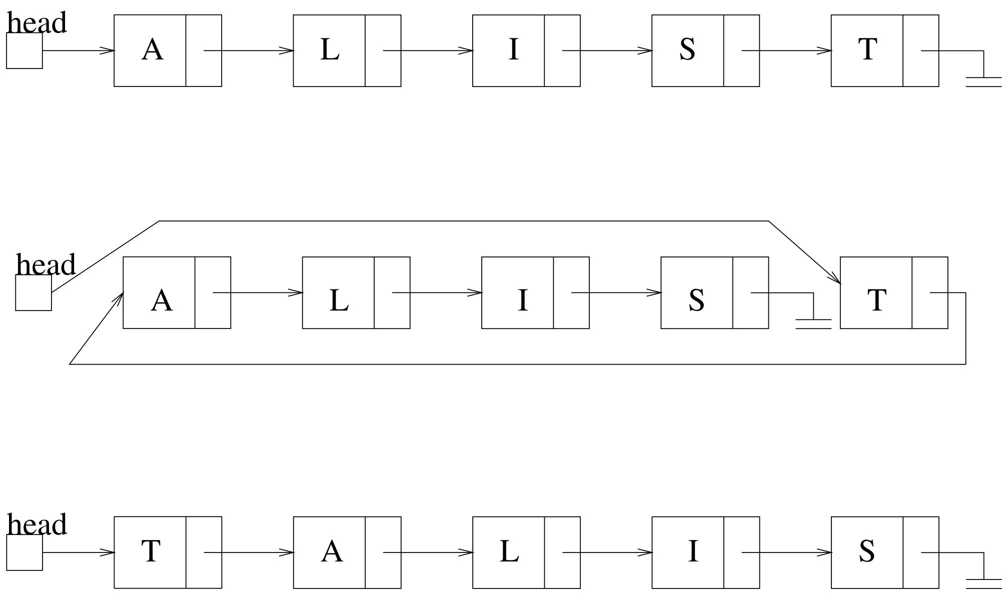
```
// C++                          // JAVA
i->next = x->next;              i.next=x.next
```



- Only one reference needs to be changed, no matter how long the list is. How does this compare with an array?
- **Note:** There is a slight problem with the C++ code. What is it?

Linked List: Moving a node

- Moving a node consist of a delete operation follow by an insert operation.
- **Example:** Move T from the end of the list to the beginning:



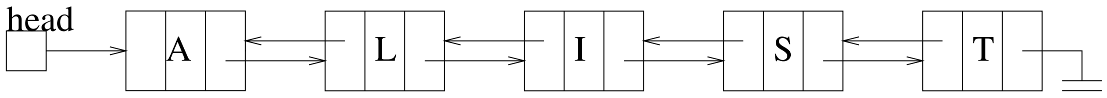
- Again, this requires changing only 3 references, no matter how long the list is. What about with arrays?

Comparison: Linked Lists and Arrays

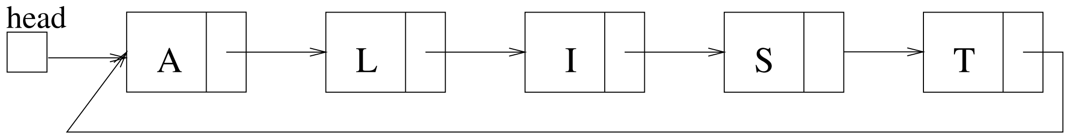
- A linked list can grow and shrink during its lifetime, and its maximum size doesn't need to be specified in advance. In contrast, arrays are always of fixed size.
- We can rearrange, add, and delete items from a linked list with only a constant number of operations. With arrays, these operations are generally linear in the size of the array.
- To find the ith entry of a linked list, we need to follow i pointers, which requires i operations. With an array, this takes only one operation.
- Similarly, it may not be obvious how large a linked list is, whereas we always know the size of an array. (This problem can be eliminated very easily. How?)

Other Linked Lists

- A **doubly linked list** is like a linked list, but each node also has a *previous* field, which points to the nodes predecessor.



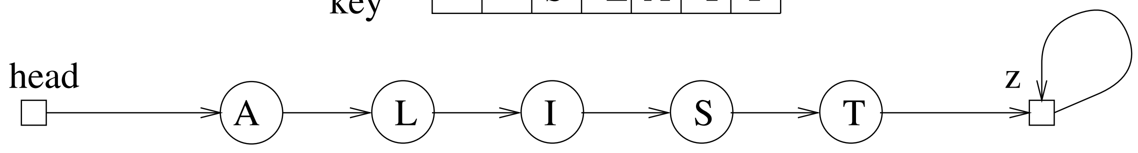
- This can simplify searching, and makes the deletion operation (potentially) easier.
- There is obviously added storage cost, and the number of instructions needed for the various operations approximately doubles.
- **Circular-linked list**:
  - The last node points to the first node.
  - It can be single or doubly linked list.
  - It can be implemented with a fixed or moving "head."



Linked Lists without Pointers

- Instead of using pointer to implement linked lists, we can use arrays.
- We won't look at this in depth, but it is not too hard to imagine how we could do it.
- There are a few complications in this type of implementation, but they can easily be worked around.
- **Example:**

next	4	1	6	5	3	2	1
key			S	L	A	I	T

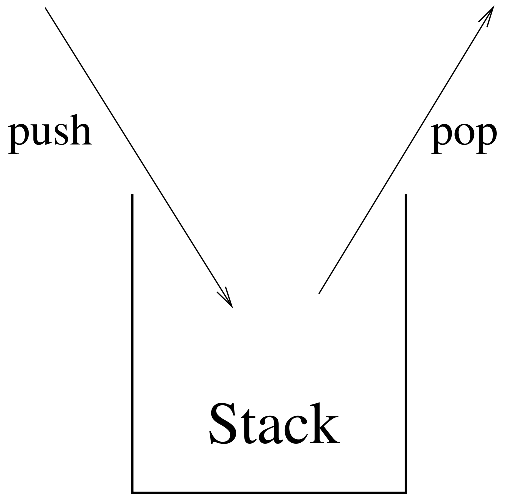


Linked Lists Summary

- Linked lists are data structures that are in many ways similar to arrays.
- Inserting, deleting or accessing items in linked lists are operations which can be performed.
- Insertion and deletion can be done in constant time.
- Finding an element in a linked list generally takes linear time. This is true whether we are trying to find an element whose value is x, or we are trying to find the ith element on the list.
- It is this last fact that can limit the usefulness of linked lists.

Stacks

- A **Stack** is a sequential organization of items in which the last element inserted is the first element removed. They are often referred to as LIFO, which stands for "last in first out."
- **Examples:** letter basket, stack of trays, stack of plates.
- The *only* element of a stack that may be accessed is the one that was most recently inserted.
- There are only two basic operations on stacks, the *push* (insert), and the *pop* (read and delete).



Stacks: Push and Pop

- The operation **push**(x) places the item x onto the top of the stack.
- The operation **pop()** removes the top item from the stack, and returns that item.
- We need some way of detecting an empty stack (This is an *underfull* stack).
  - In some cases, we can have **pop()** return some value that couldn't possibly be on the stack.
  - **Example:** If the items on the stack are positive integers, we can return "-1" in case of underflow.
  - In other cases, we may be better off simply keeping track of the size of the stack.
- In some cases, we will also have to worry about filling the stack (called *overflow*). One way to do this is to have **push**(x) return "1" if it is successful, and "0" if it fails.

An Example Stack Operations

Assume we have a stack of size 3 which holds integers between -100 and 100. Here is a series of operations, and the results.

Operation	Stack Contents	Return
create	()	
push(55)	(55)	1
push(-7)	(-7,55)	1
push(16)	(16,-7,55)	1
pop	(-7,55)	16
push(-8)	(-8,-7,55)	1
push(23)	(-8,-7,55)	0
pop	(-7,55)	-8
pop	(55)	-7
pop	()	55
pop	()	101

Implementing Stacks: Array

- Stacks can be implemented with an array and an integer top that stores the array index of the top of the stack.
- Empty stack has $top = -1$, and a full stack has $top = n - 1$, where n is the size of the array.
- To push, increment the top counter, and write in the array position.
- To pop, decrement the top counter.

Example of Array Implementation of Stack

- We use an array $E[0..4]$ to store the elements and a variable p to keep track of the top. The last column, labeled “R” is the result of the function call.

Operation	p	E0	E1	E2	E3	E4	R
create	-1	?	?	?	?	?	
push(55)	0	55	?	?	?	?	1
push(-7)	1	55	-7	?	?	?	1
push(16)	2	55	-7	16	?	?	1
pop	1	55	-7	16	?	?	16
push(-8)	2	55	-7	-8	?	?	1
pop	1	55	-7	-8	?	?	-8
pop	0	55	-7	-8	?	?	-7

- Notice that some values are still in the array, but are no longer considered to be in the stack. In general, elements $E[i]$ are “garbage” if $i > p$. Why don’t we erase the element (i.e. set it to some default value.)?

Example Application of Stacks

- Stacks can be used to check a program for balanced symbols (such as {},(),[ ]).
- **Example:** {()} is legal, as is {()({})}, whereas {((} and {(}) are not (so simply counting symbols does not work).
- If the symbols are balanced correctly, then when a closing symbol is seen, it should match the "most recently seen" unclosed opening symbol. Therefore, a stack will be appropriate.

The following algorithm will do the trick:

- While there is still input:

```
$s$ = next symbol
if ($s$ is an opening symbol) push($s$)
else                //$s$ is a closing symbol
    if (Stack.Empty) report an error
    else $r$ = pop()
    if(! Match($s$,$r$)) report an error
```

- If (! Stack.Empty) report an error

Examples

1. Input: { ( ) }
   - Read {, so push {
   - Read (, so push (. Stack has { (
   - Read), so pop. popped item is (which matches). Stack has now {.
   - Read }, so pop; popped item is { which matches }.
   - End of file; stack is empty, so the string is valid.

2. Input: { ( ) ({) } } (This will fail.)

2. Input: { ({ }){ } ( ) } (This will succeed.)

3. Input: { ( ) }) (This will fail.)

Stack: C++ Array Implementation

- The following is a C++ stack that holds items of type *ItemType*

```
class Stack
  {
   private:
     ItemType *stack;
     int p;
   public:
     Stack(int max=100)
       {stack = new ItemType[max];
        p = 0; }
    ~Stack()
       {delete stack; }
     void push(ItemType v)
       { stack[p++] = v; }
     ItemType pop()
       {return stack[--p]; }
     int empty()
       {return !p; }
   };
```

Stack: JAVA Array Implementation

- The following is a JAVA stack that holds items of type *ItemType*

```
public class Stack {
     private ItemType[] stack;
     private int p;

     public Stack(int max) {
        stack = new ItemType[max];
        p = 0;
     }
     public void push(ItemType v) {
        stack[p++] = v;
     }
     public ItemType pop() {
        return stack[--p];
     }
     public int empty() {
        return !p;
     }
   };
```

Operator Precedence Parsing

- We can use the stack class we just defined to parse and evaluate mathematical expressions like:

$$5 * (((9 + 8) * (4 * 6)) + 7)$$

- First, we transform it to postfix notation (How would you do this?):

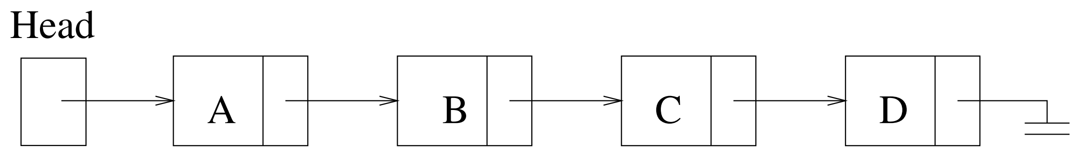
$$5\ 9\ 8 + 4\ 6 * * 7 + *$$

- Then, the following C++ routine uses a stack to perform this evaluation:

```
char c; Stack acc(50); int x;
while (cin.get(c))
  {
   x = 0;
   while (c == ' ' ) cin.get(c);
   if (c == '+') x = acc.pop() + acc.pop();
   if (c == '*') x = acc.pop() * acc.pop();
   while (c>='0' && c<='9')
     {x = 10*x + (c-'0'); cin.get(c); }
   acc.push(x);
  }
cout << acc.pop() << '\n';
```

Stack Implementation: C++ Linked Lists

- We can use linked lists to implement stacks.
- The head of the list represents the top of the stack.
- **Example** After $push(D)$, $push(C)$, $push(B)$, $push(A)$, we have:



-
  ```
  ItemType pop () {
       ItemType x = head->key;
       head=head->next;
       return x; }
  ```

-
  ```
  void push(ItemType x) {
       node *x;
       x=new node;
       x->key = X;
       insert(x); }
  ```
- **Note:** Slight error in `pop`. What is it?

Stack Implementation: JAVA Linked Lists

We assume *head* points to the first element of the list, and is the top of the stack.

-
  ```
  public ItemType pop () {
      ItemType x = head.key;
      head=head.next;
      return x;
  }
  ```

-
  ```
  public void push(ItemType x) {
      node A=new node();
      A.key = x;
      A.next=head;
      head=A;
  }
  ```

Stack Implementation: Array or Linked List?

Linked Lists

- Use 1 pointer extra memory per item. If an item is an integer, that means twice as much space is used. If an item is a structure/class consisting of many objects, it is only a small price to pay.
- Are unlimited in size.

Arrays

- Allocate a constant amount of space, some of which may never be used. The amount of wasted memory is the number of unused elements times the size of the item, which could be large in some cases.
- The maximum size is determined when the stack is created.

Which is better? Why?

Stack Applications

- Recursion removal can be done with stacks.
- Reversing things is easily done with stacks.
- Procedure call and procedure return is similar to matching symbols:
  - When a procedure returns, it returns to the most recently active procedure.
  - When a procedure call is made, save current state on the stack. On return, restore the state by popping the stack.

Queues

- A **Queue** is a sequential organization of items in which first element entered is first removed. They are often referred to as FIFO, which stands for “first in first out.”
- **Examples:** standing in a line, printer queue.
- The basic operations are:
  - *insert(x)* places x at the beginning of the queue.
  - *remove()* returns and deletes the item at the end of the queue.
- **Example:**

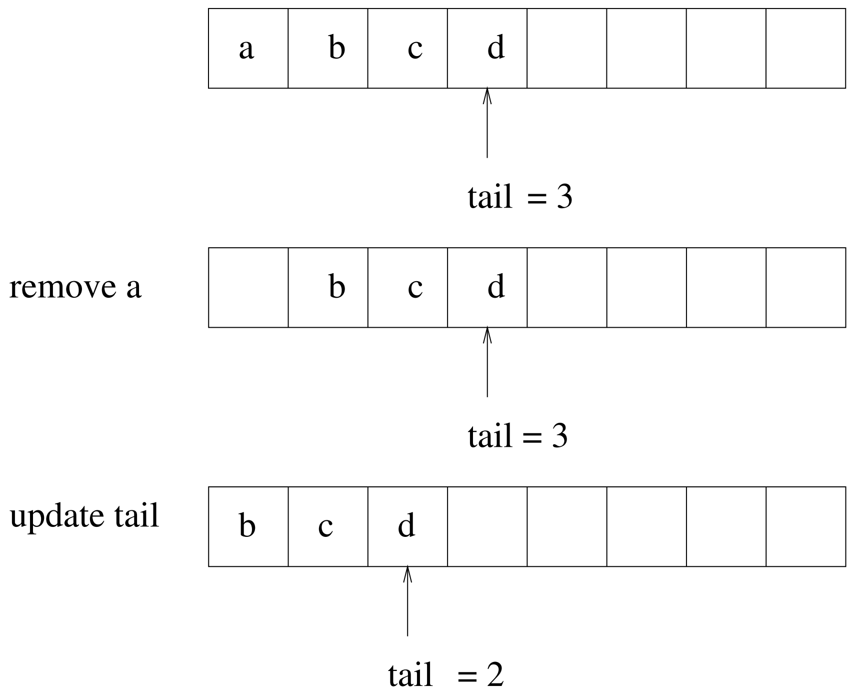
Operation	Queue	Return
CreateQueue	()	
insert(7)	(7)	
insert(8)	(7,8)	
insert(5)	(7,8,5)	
remove()	(8,5)	7
remove()	(5)	8

Queue Applications

- Operating systems:
  - Queue of jobs or processes ready to run (waiting for CPU):
  - Queues of processes waiting for I/O.
  - Files sent to printer
- Simulation of real-world queuing systems:
  - Customers in a grocery store, bank, etc.
  - Orders in a factory
  - Hospital emergency room or doctor's office
  - Telephone calls for airline reservations, customer orders, information, etc.
- Problem applications:
  - Topological ordering: given a sequence of $events$, and pairs (a, b) indicating that event a should occur prior to b, provide a schedule.

Queues: Naive Implementation

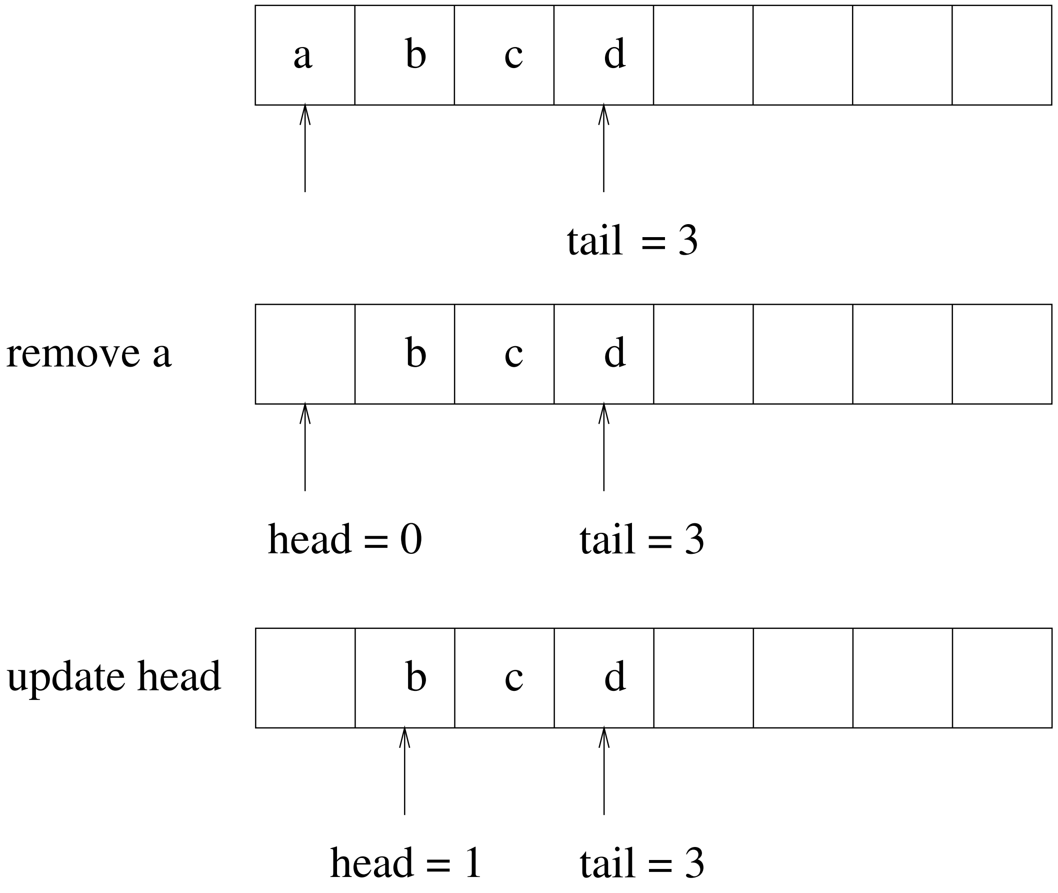
- Using array:
  - Store items in an array. The head it the first element, and the tail is indicated by a variable *tail*.
  - *insert*(x) is easy: increment $tail$, and insert element.
  - *remove*() is inefficient: all elements have to be shifted. Thus, *remove* is $O(n)$.



- How can we improve this?

Queues: A Better Implementation

- Keep track of both the *head* and the *tail*.
- To remove, increment $front$.



- There is still a problem. What is it, and how can we fix it?

Queues: Circular Array Implementation

- Previous implementation is $O(1)$ per operation, which is great.
- However, after n inserts (where n is the size of the array), the array is full even if the queue is logically nearly empty.
- **Solution:** Use wraparound to reuse the cells at the start of the array. To $increment$, add one, but if that goes past end, reset to zero.

- How do you detect a *full* or *empty* queue?
- We will give a simplified implementation for the queue data structure. A better implementation would detect an empty (full) queue before performing a dequeue (enqueue) operation.

Queue C++ Declaration

- Here is a C++ declaration of an integer queue using a “circular” array:

```
class Queue {
private:
  int *queue;
  int cap,head,tail;
public:
Queue(int s=100) {
      cap= s;
      queue = new int[cap];
      head = 0; tail = 0; }
~Queue() { delete queue; }
void Enqueue(int v) {
        queue[tail] = v;
        tail = (tail+1) % cap;
     }
int Dequeue() {
       int t = queue[head];
       head = (head + 1) % cap;
       return t;
    }
int Empty() {return (head == tail ); }
int Full() {return (head == ((tail+1)%cap)); }
};
```

Queue JAVA Declaration

```
public class Queue {
  private int[] queue;
  private int cap,head,tail;
public Queue(int s) {
      cap=s;
      queue = new int[cap];
      head = 0;
      tail = 0; i
      }
public void Enqueue(int v) {
      queue[tail] = v;
      tail = (tail+1) % cap;
      }
public int Dequeue() {
      int t = queue[head];
      head = (head + 1) % cap;
      return t;
      }
public int Empty() {
      return (head == tail ); }
public int Full() {
      return (head == ((tail+1)%cap)); }
};
```

Example of Queue Using Circular Array

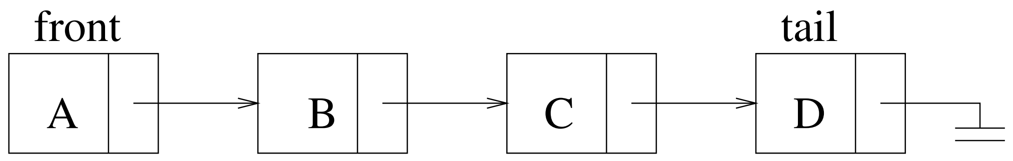
- Here we use an array $E[0..3]$ to store the elements and the variables h (head) and t (tail) to keep track of the beginning and end of the queue. The value *R* is the return value of the operation.

Operation	h	t	E0	E1	E2	E3	R
create	0	0	?	?	?	?	
insert(55)	0	1	55	?	?	?	
insert(-7)	0	2	55	-7	?	?	
insert(16)	0	3	55	-7	16	?	
remove()	1	3	55	-7	16	?	55
insert(-8)	1	0	55	-7	16	-8	
remove()	2	0	55	-7	16	-8	-7
remove()	3	0	55	-7	16	-8	16
insert(11)	3	1	11	-7	16	-8	

- Note that some of the values remain physically in the array, but are logically no longer in the queue.

Queues: Linked List Implementation

- We can maintain $front$ and $tail$ pointers.
- $remove$: advance $front$.
- $insert$: add to end of list and adjust $tail$.



- The details are not very hard to work out, so will not be presented.