Stack

(LIFO ordering structure)

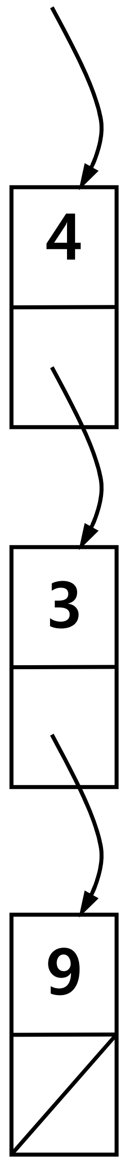
Stack ADT

```
template <class T>
class Stack {
public:
    Stack();

    bool IsEmpty() const;

    // T& Top() const;

    void Push(const T& data);

    T Pop();

private:
     Not yet!!

};
```

```
int main() {
     List<int> mystack;
     mystack.Push(9);
     mystack.Push(3);
     mystack.Push(2);
     mystack.Pop();
     mystack.Push(4);
      . . . .
}
```

Linked list based implementation

```
template <class T>
class Stack {
public:
   Stack();

   bool IsEmpty() const;

   void Push(const T& d);

   T Pop();

private:
    struct stackNode {
         T data;
         stackNode *next;
         stackNode();
    };
    stackNode *top;
    int size;
};
```



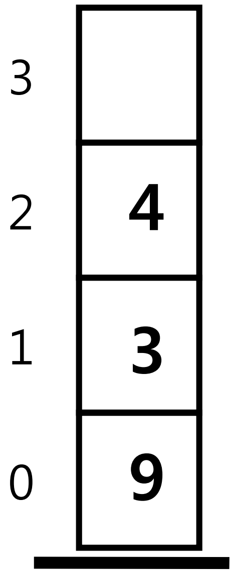

```
template<class T>
void Stack<T>::Push(const T & d)
{
    stackNode *newNode =
                 new stackNode(d);

     newNode->next = top;
     top = newNode;
     size++;
}
```

```
// Assume stack is not empty
template<class T>
T Stack<T>::Pop() {

}
```

Array based implementation

```
template <class T>
class Stack {
public:
    Stack();

    bool IsEmpty() const;

    void Push(const T& d);

    T Pop();

private:
     int capacity;
     int size;
     T *stack;
};
```

```
template <class T>
void Stack<T>Push(const T &d) {
    if (size == capacity) {
        // resize stack array
    }
    stack[size] = d;
    size++;
}
```

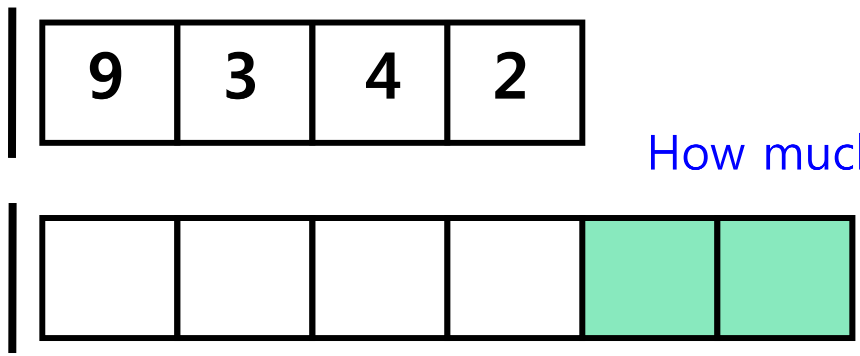


size = 3

```
template <class T>
Stack<T>::Stack() {
    capacity = 4;
    size = 0;
    stack = new T[capacity];
}
```

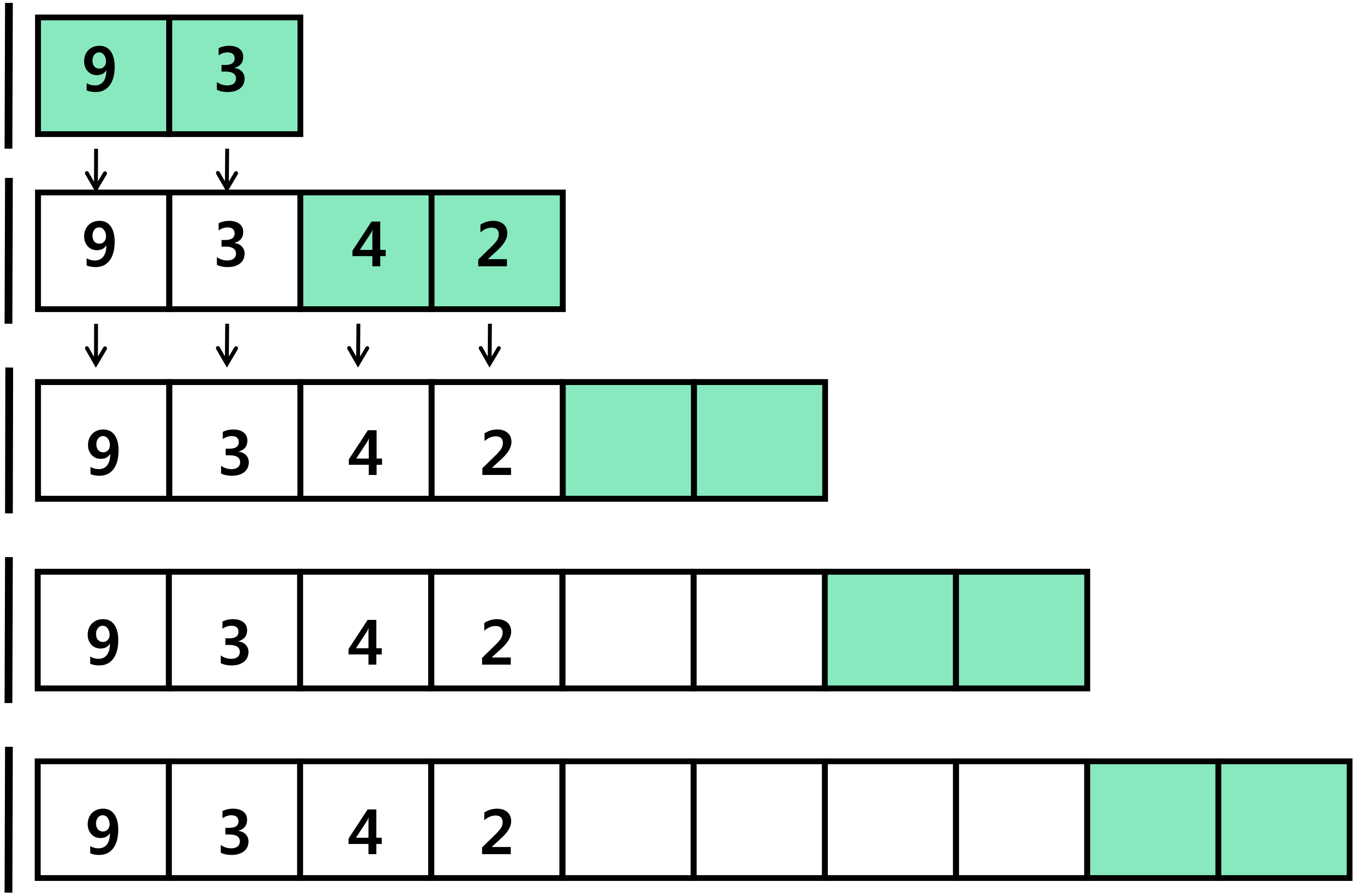
Array resizing when inserting (pushing), but the array is full

Steps.



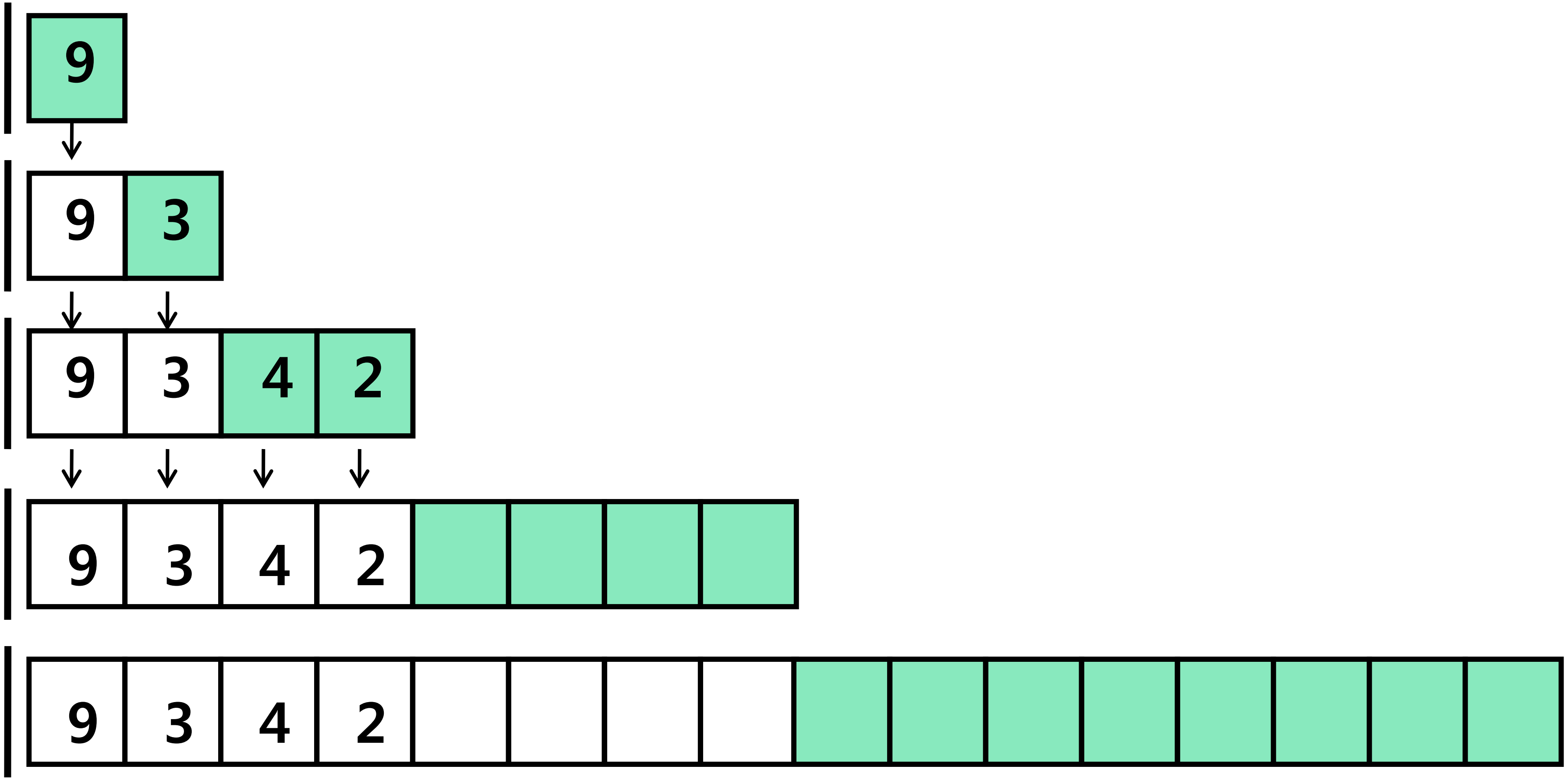
How much do we need to increase?

Strategy 1: Increase array size by a constant amount



How much is the cost by this strategy on a sequence of n pushes?

Strategy 2: Double the array size



How much is the cost by this strategy on a sequence of n pushes?

Run time summary

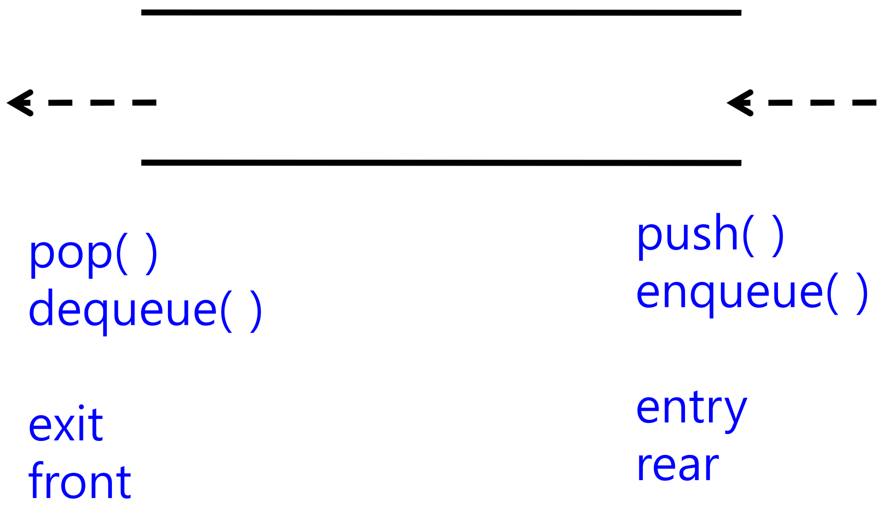
- Linked list based implementation
  - O( ) per push, pop
- Array based implementation
  - O( ) per pop
  - O( ) per push if capacity exists.
  - Cost over n pushes is O( ) for an average of O( ) per push.
- Which implementation we choose?

Queue

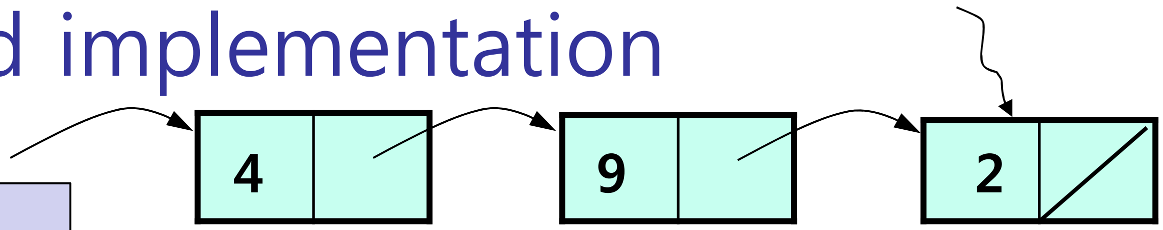
(FOFO ordering structure)

Queue ADT

```
template <class T>
class Queue
{
public:
   Queue ();
   Boolean IsEmpty() const;

   // T& Front() const ;
   // T& Rear() const;
   void Push(const T& item);
   T Pop();

Private:
    // not yet
   };
```



Linked list based implementation




```
template <class T>
class Queue {
public:
   Queue();

   bool IsEmpty() const;
   void Push(const T& d);
   T Pop();

private:







};
```

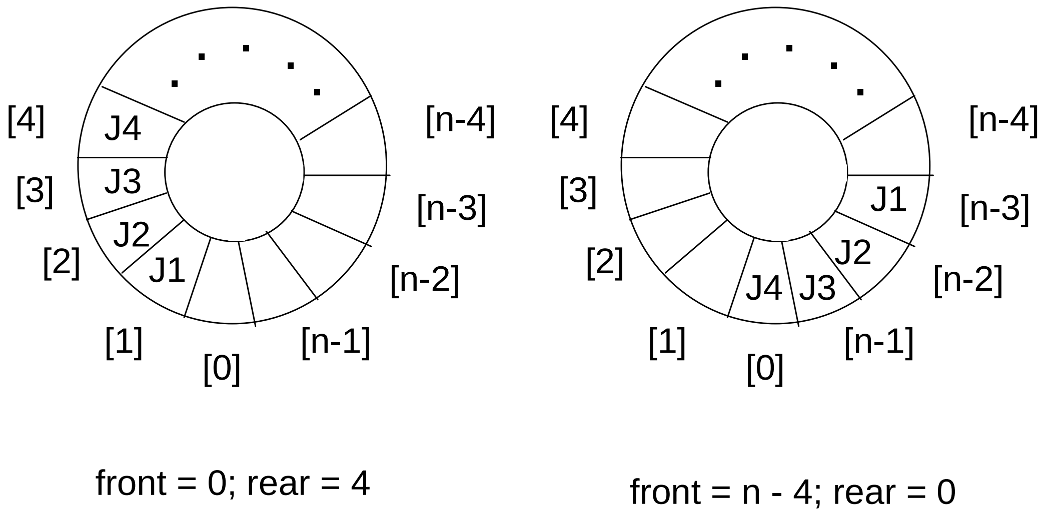
Which pointer is "entry" ("rear") and which is "exit" ("front")?

Running time of Push?

Running time of Pop?

Array based implementation

```
template <class T>
class Queue {
public:
   Queue();

   bool IsEmpty() const;
   void Push(const T& d);
   T Pop();

private:
    T *queue;
    int capacity;
    int front
    int rear;
};
```



Modeling as a circular list

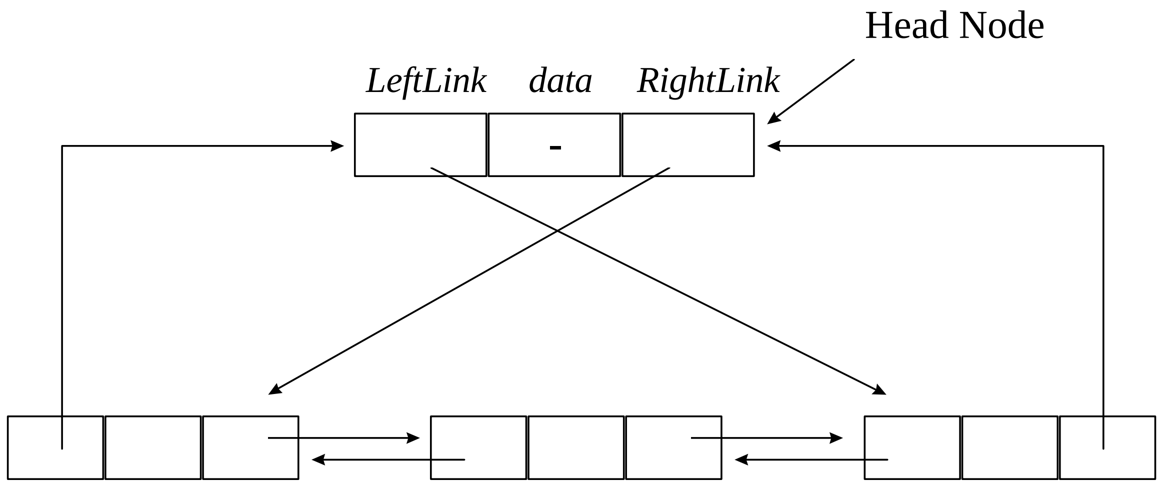
DLL

(Doubly Linked Lists)

- Problems with singly linked lists
  - can move only in the direction of the links
  - deletion requires knowing the preceding node
- Node
  - at least three fields :

    data, llink (left link), rlink (right link)

DLL with sentinel node



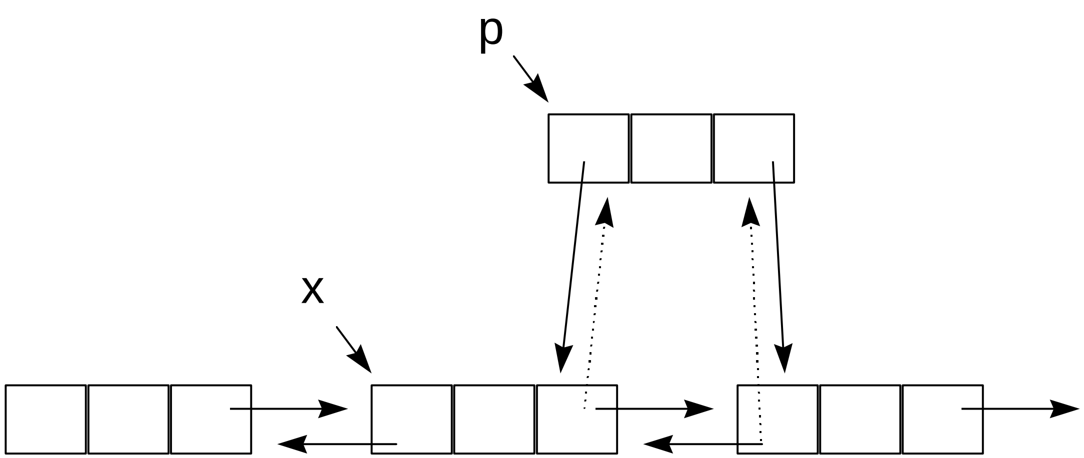
- Head node
  convenient for algorithms
  data field usually contains no information
- Essential virtue
  one can go back and forth with equal ease
  p==p→llink→rlink==p→rlink→llink

Implementation

```
class DblList;
class DblListNode {
// Dblist functions are allowed to access
friend class DblList;
private:
   int data;
   DblListNode *left, *right;
};

class DblList {
public:
  // List manipulation operations
    ⋮
private:
   DblListNode *first; // points to head node
};
```

Insert a node

```
void DblList::Insert(DblListNode *p, DblListNode *x)
// insert node p to the right of node x
{
   p->left = x; p->right = x->right;
   x->right->left = p; x->right = p;
}
```



Delete a node

```
void DblList::Delete(DblListNode *x)
{
   if ( x == first ) throw "Deletion of headnode not permitted“;
   else {
      x->left->right = x->right;
      x->right->left = x->left;
      delete x;
   }
}
```