Basic Data Structures

- Abstract Data Type
- Stack & Queue
- Linked List

Prof. Ki-Hoon Lee
School of Computer and Information Engineering
Kwangwoon University

Abstract Data Type

Data Abstraction

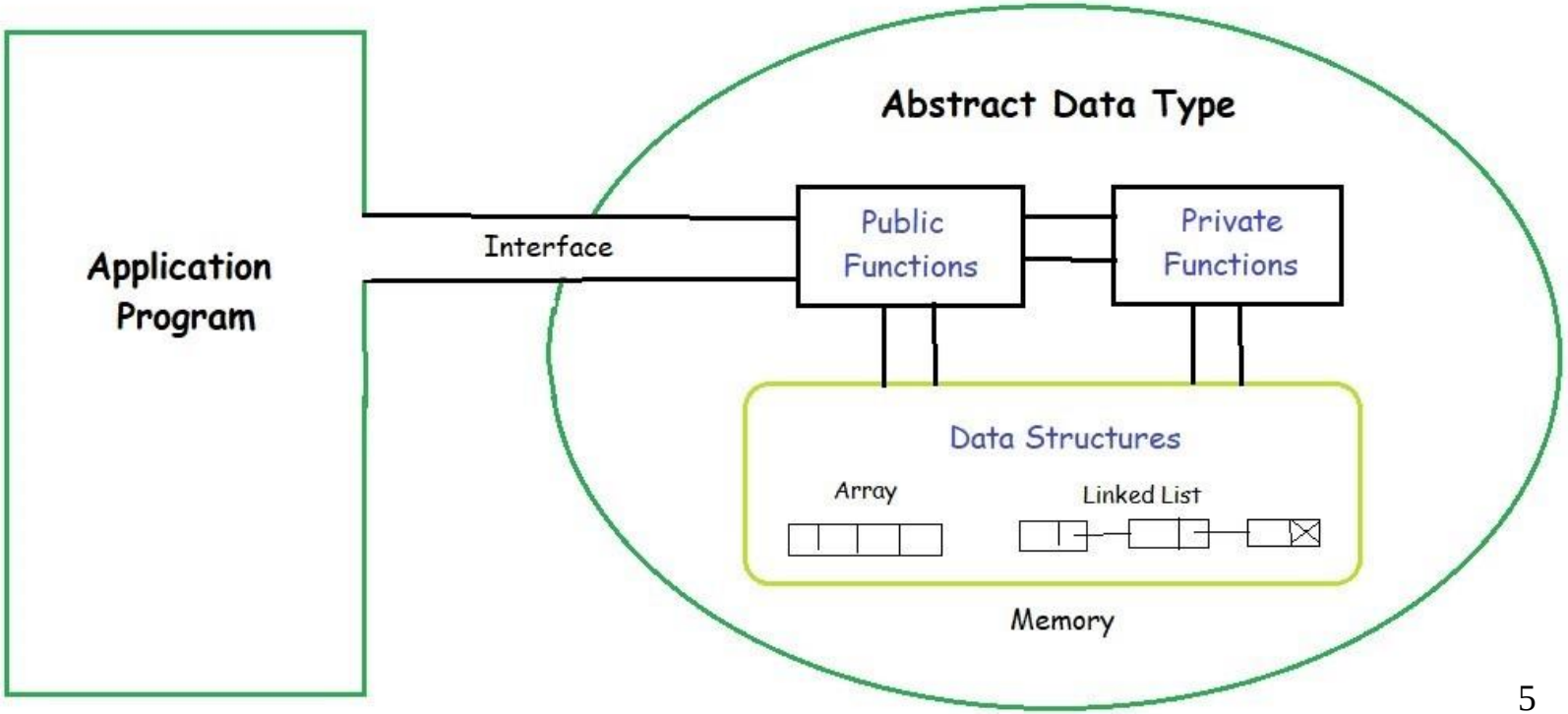
- Making a clear distinction between *what* the object does and *how* the object does it.
- The separation between the *specification* of an object and its *implementation*
- Example: Stack
  - Specification: push, pop, …
  - Implementation: array or linked list

Abstract Data Type

- Data Type
  - A collection of objects (e.g., integers) and a set of operations (e.g., +, –, *) that act on those objects
- Abstract Data Type (ADT)
  - A data type that is organized in such a way that the specification is separated from its implementation.
  - e.g.) C++ Class

Abstract Data Types in C++

- C++ allows the data and operations of an ADT to be defined together.
- It also enables an ADT to prevent access to internal implementation details.



https://www.geeksforgeeks.org/abstract-data-types/

Data Structure

- Data structures serve as the basis for abstract data types (ADT).
  - The ADT defines the logical form of the data type.
  - The data structure implements the physical form of the data type.

- Data structures provide a means to manage large amounts of data efficiently.

C++ Class

- Constructor
  - A special member function that is executed whenever we create new objects of that class
  - A constructor has the same name as the class, and it does not have any return type at all, not even void.
  - Constructors can be very useful for setting initial values for certain member variables.

C++ Class

- Destructor
  - A special member function that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied
  - A destructor has the same name as the class prefixed with a tilde (~), and it can neither return a value nor can it take any parameters.
  - Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories, and so on.

C++ Class

- Template
  - The foundation of generic programming, which involves writing code in a way that is independent of any particular type.
  - The simple idea is to pass data type as a parameter so that we don’t need to write the same code for different data types.

C++ Template Class

```
#include <iostream>
using namespace std;

// class template declaration part
template <class T>
class Test
{
    public:
        // constructor
        Test();

        // destructor
        ~Test();

        // method
        T Data(T);
};
```

C++ Template Class

```
// constructor
template <T>
Test<T>::Test()
{cout << "Constructor, allocate..." << endl;}

// destructor
template <T>
Test<T>::~Test()
{cout << "Destructor, deallocate..." << endl;}

// method
template <T>
T Test<T>::Data(T v)
{return v;}
```

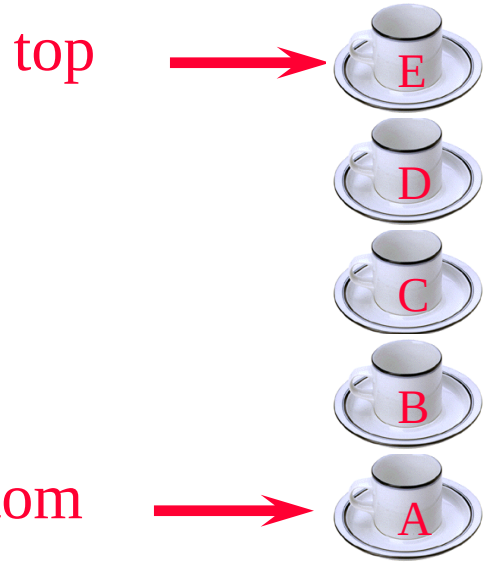
```
// the main program
int main(void)
{
    Test<int> Var1;
    Test<double> Var2;
    Test<char> Var3;

    cout << Var1.Data(100) << endl;
    cout << Var2.Data(1.234) << endl;
    cout << Var3.Data('K') << endl;
    return 0;
}
```

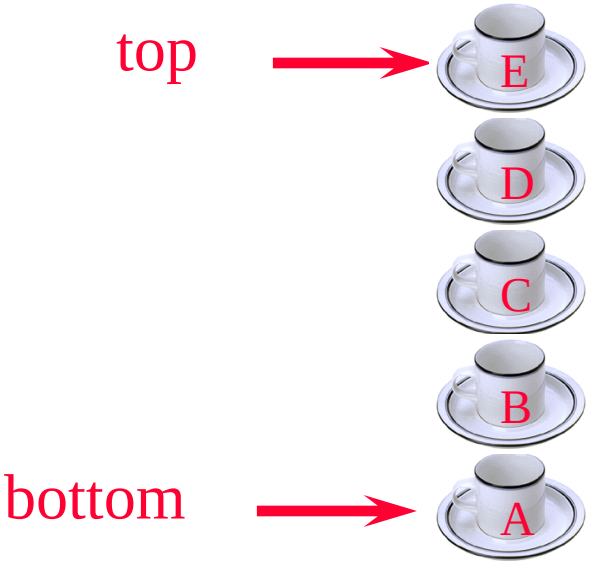
Stack & Queue

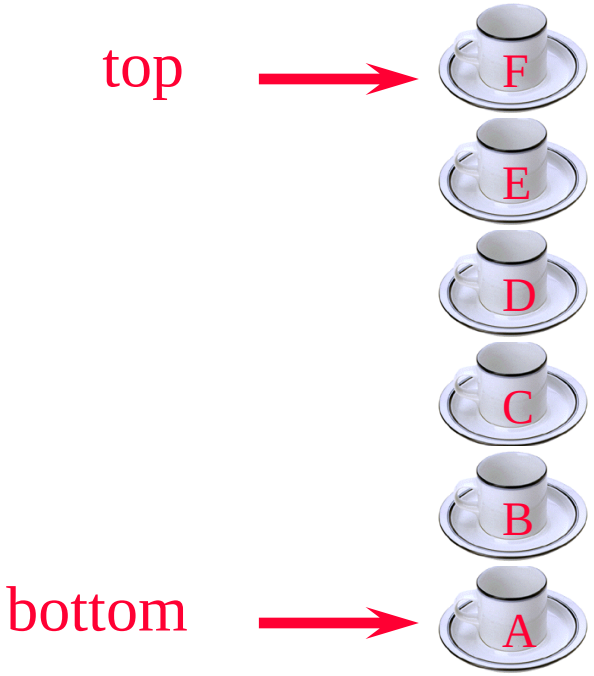
Stacks

- A LIFO (Last-In-First-Out) list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.



Stack of Cups



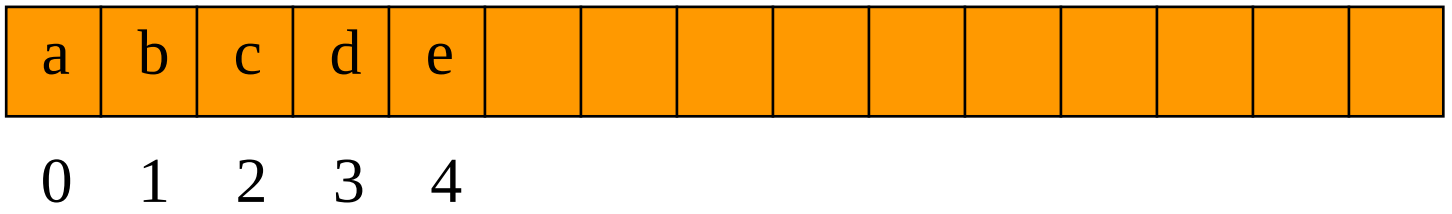


Stacks

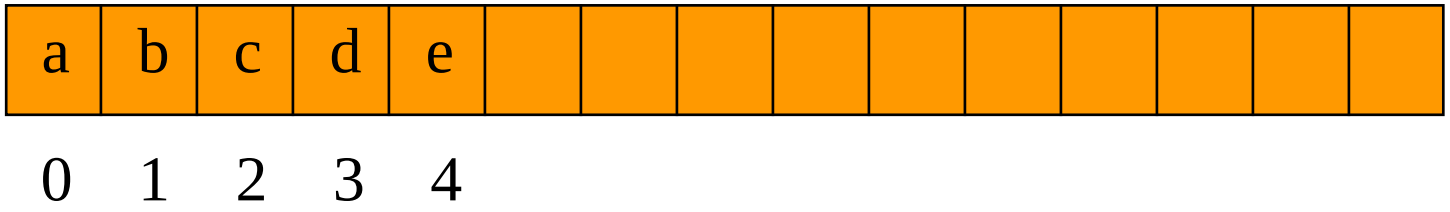
- Standard operations:
  - IsEmpty … returns true iff stack is empty
  - Top … returns top element of stack
  - Push … adds an element to the top of the stack
  - Pop … deletes the top element of the stack

Stacks

- Use a 1D array to represent a stack.
- Stack elements are stored in stack[0] through stack[top].

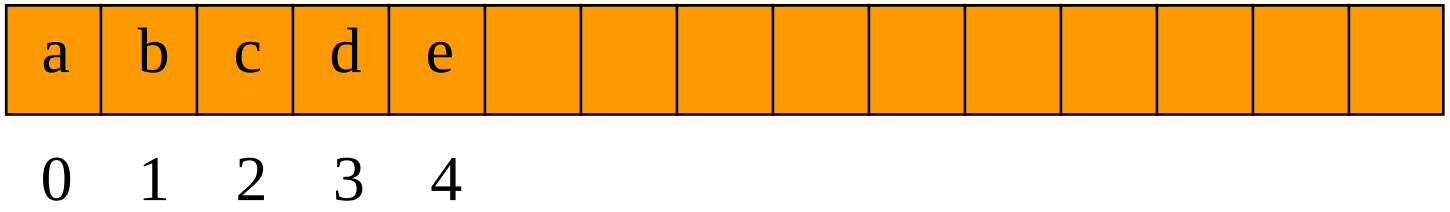


Stacks



- stack top is at element e
- IsEmpty() => check whether top >= 0
  - O(1) time
- Top() => If not empty return stack[top]
  - O(1) time

Derive From arrayList



- Push(theElement) => if array full (top == capacity – 1) increase capacity and then add at stack[top+1]
  - O(capacity) time when full; otherwise O(1)

- pop() => if not empty, delete from stack[top]
  - O(1) time

```
template<class T>
class Stack
{
   public:
    Stack(int stackCapacity = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const;
    T& Top() const;
    void Push(const T& item);
    void Pop();
   private:
    T *stack;      // array for stack elements
    int top;       // position of top element
    int capacity; // capacity of stack array
};
```

Constructor

```
template<class T>
Stack<T>::Stack(int stackCapacity)
                :capacity(stackCapacity)
{
   if (capacity < 1)
     throw "Stack capacity must be > 0";
   stack = new T[capacity];
   top = -1;
}
```

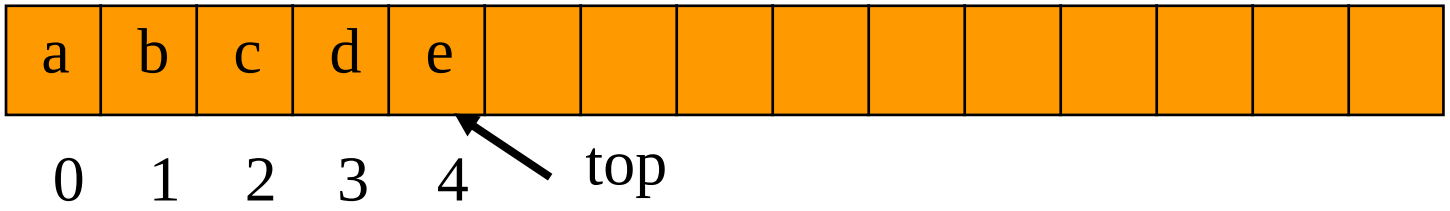
IsEmpty

```
template<class T>
inline bool Stack<T>::IsEmpty() const
   {return top == -1}
```

Top

```
template<class T>
inline T& Stack<T>::Top() const
{
    if (IsEmpty())
        throw "Stack is empty";
    return stack[top];
}
```

Push




```
template<class T>
void Stack<T>::Push(const T& x)
{// Add x to the stack.
   if (top == capacity - 1)
      {ChangeSize1D(stack, capacity,
                              2*capacity);
       capacity *= 2;
      }
   // add at stack top
   stack[++top] = x;
}
```

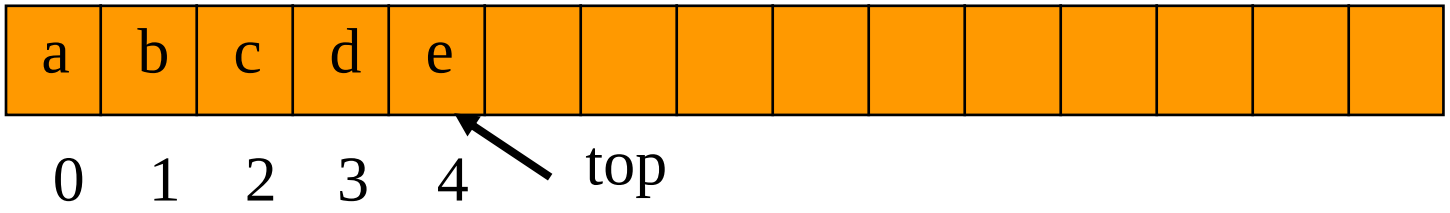
```
#define _CRT_SECURE_NO_WARNINGS
#include <algorithm>
using namespace std;
```

```
template <class T>
void ChangeSize1D(T*& a, const int oldSize, const int newSize)
{
  if (newSize < 0) throw "New length must be >= 0";

  T* temp = new T[newSize];              // new array
  int number = min(oldSize, newSize);    // number to copy
  copy(a, a + number, temp);
  delete [] a;                           // deallocate old memory
  a = temp;
}
```

http://www.cplusplus.com/reference/algorithm/copy/

Pop




```
void Stack<T>::Pop()
{
    if (IsEmpty())
        throw "Stack is empty. Cannot delete.";
    stack[top--].~T(); // destructor for T
}
```

Parentheses Matching

- ((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

- Scan expression from left to right.
- When a left parenthesis is encountered, add its position to the stack.
- When a right parenthesis is encountered, remove matching position from stack.

Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13)

Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19)

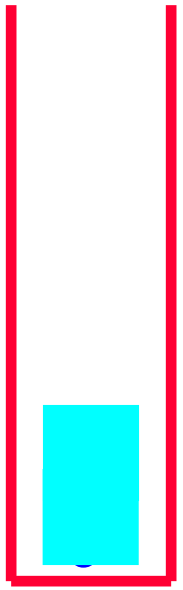
Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19) (21,25)

Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

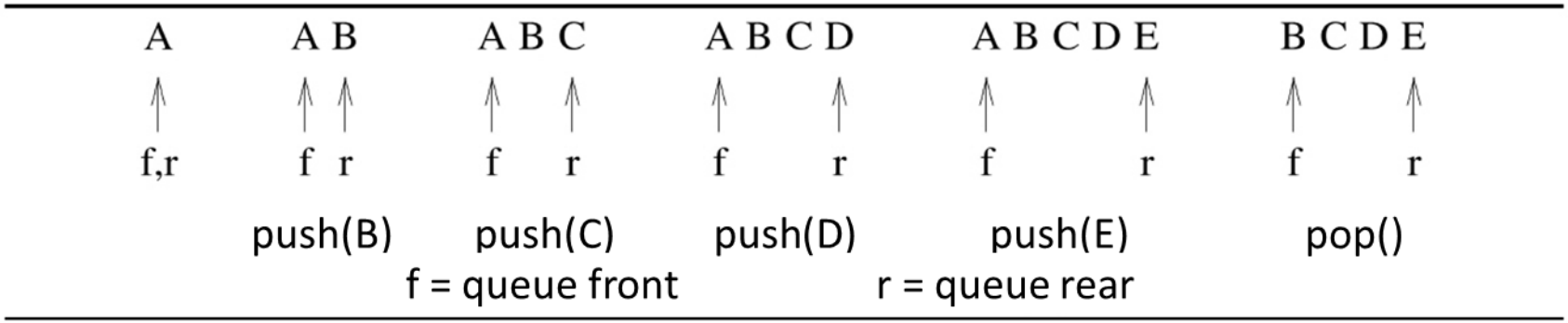


(2,6) (1,13) (15,19) (21,25)(27,31) (0,32)

- and so on

Queues

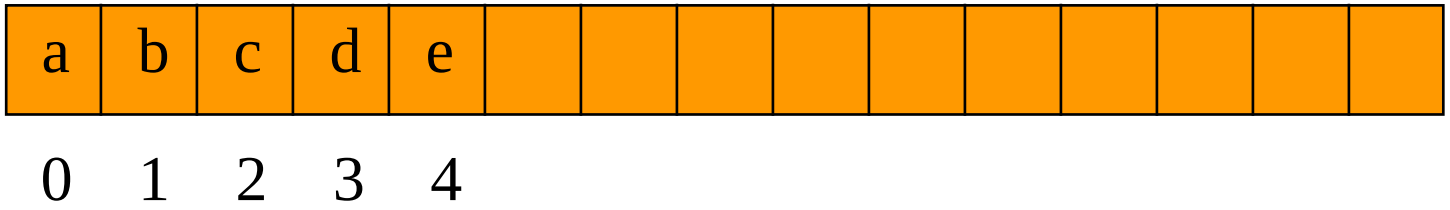
- A FIFO (First-In-First-Out) list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.



Queue Operations

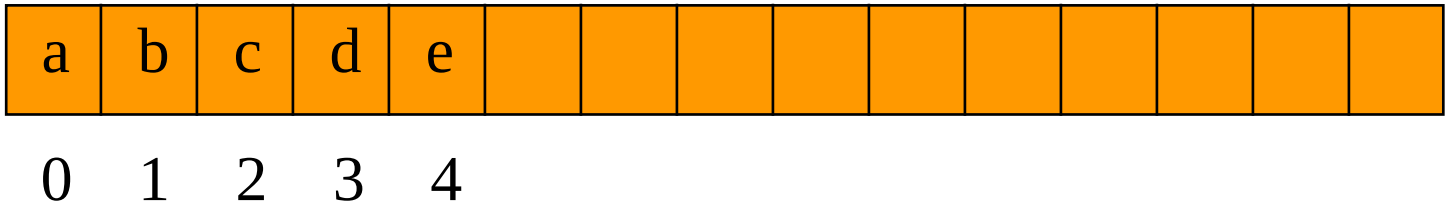
- IsEmpty … returns true iff queue is empty
- Front … returns front element of queue
- Rear … returns rear element of queue
- Push … adds an element at the rear of the queue
- Pop … deletes the front element of the queue

Queue in an Array



- Use a 1D array to represent a queue.
- Suppose queue elements are stored with the front element in queue[0], the next in queue[1], and so on.

Derive From arrayList

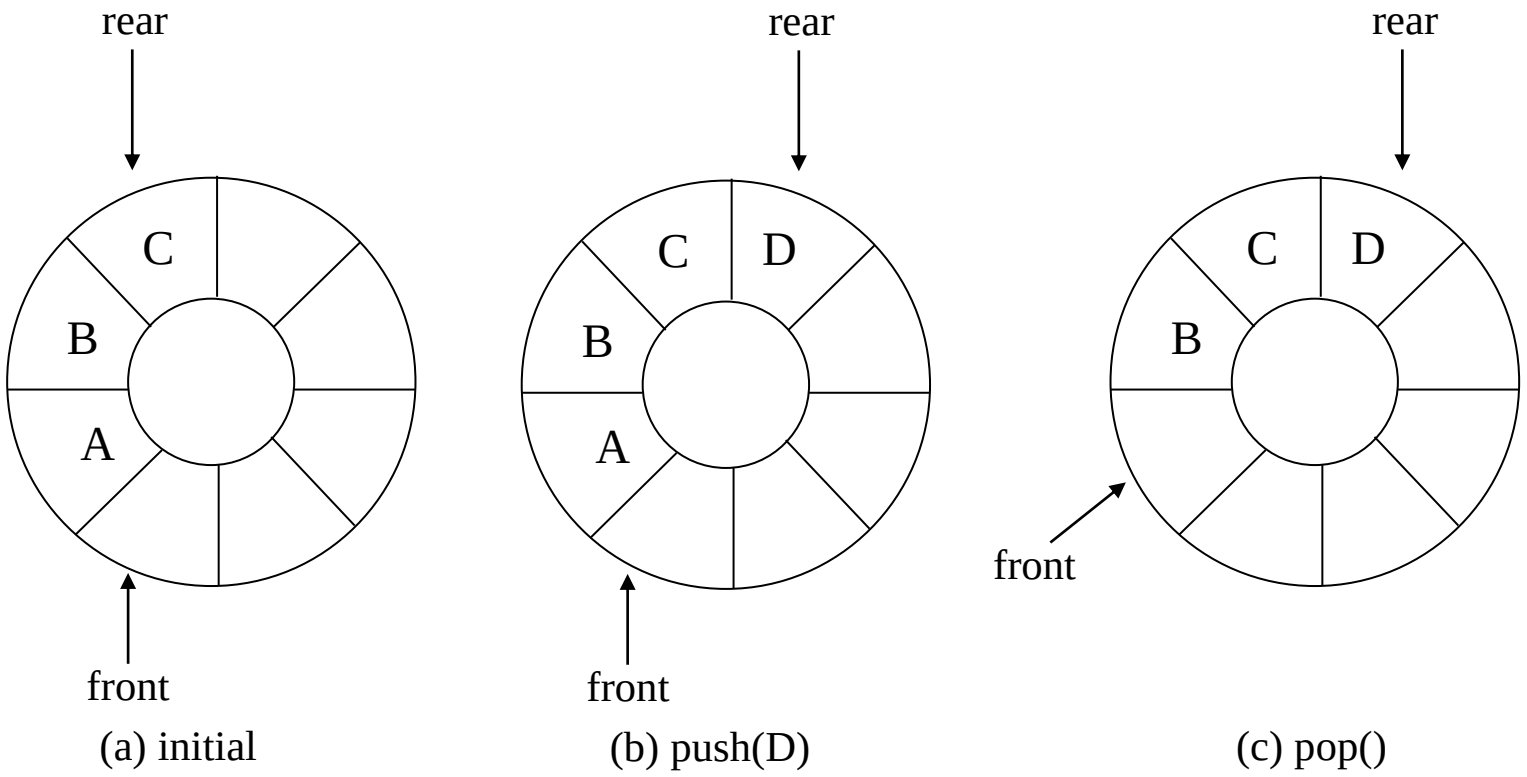


- Pop() => delete queue[0]
  - – shift the remaining elements to the left
  - – O(queue size) time

- Push(x) => if there is capacity, add at right end
  - – O(1) time (excluding array doubling)

Circular Queue

- To perform pop and push in O(1), we use a circular representation.

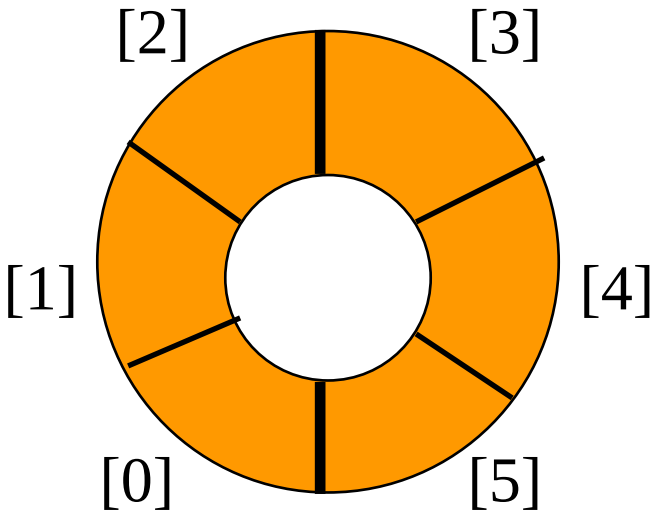


Circular Queue

- Use a 1D array queue.

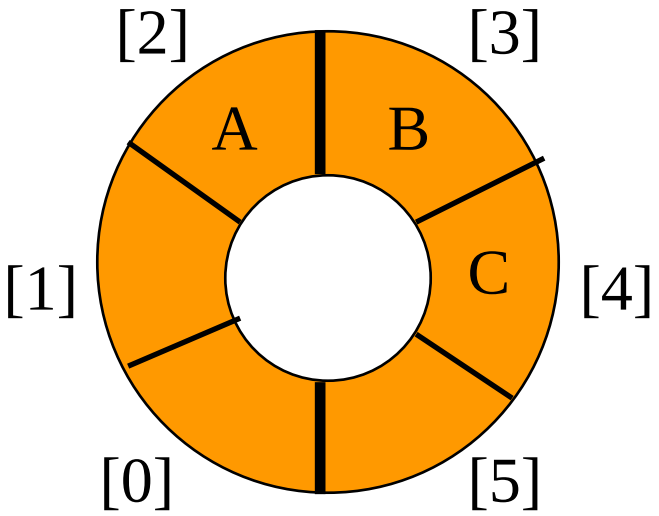
queue[]

- Circular view of array.



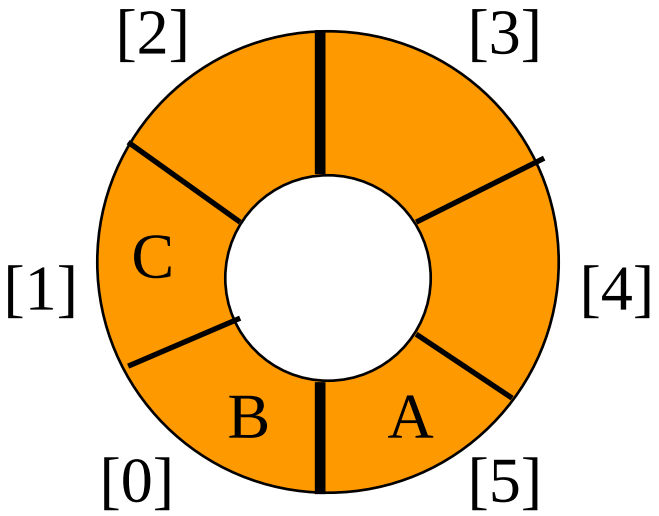
Circular Queue

- Possible configuration with 3 elements.



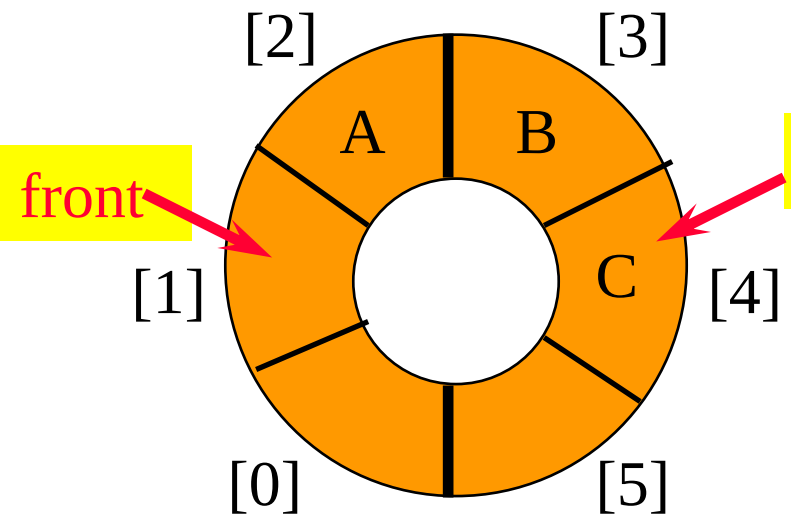
Circular Queue

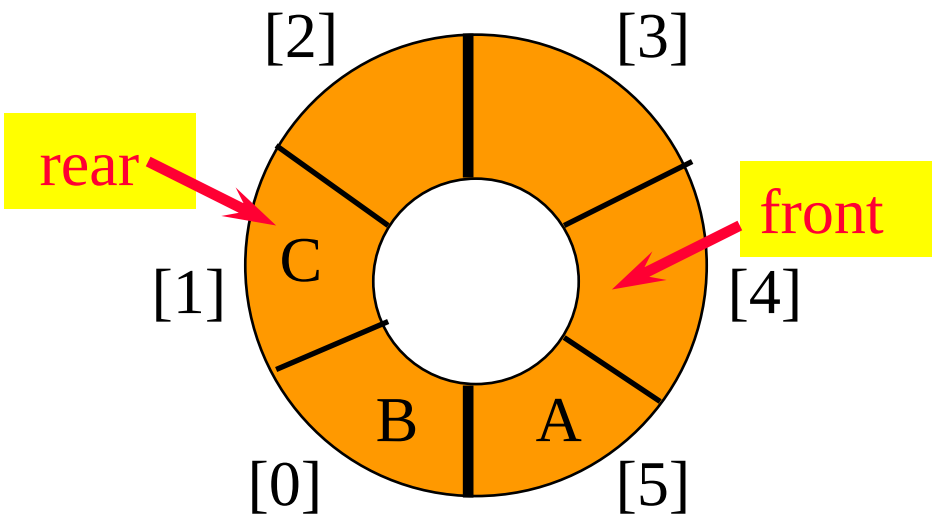
- Another possible configuration with 3 elements.



Circular Queue

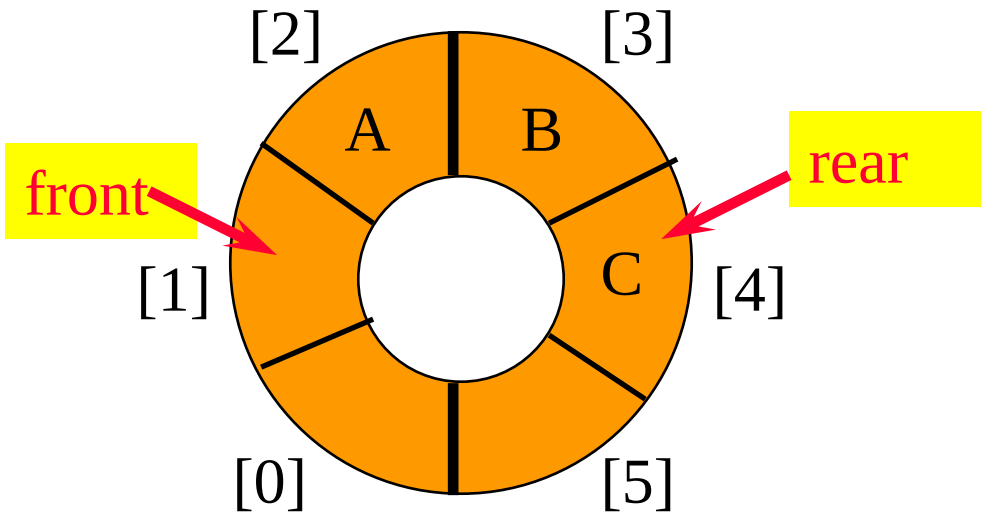
- Use integer variables front and rear.
  - Front is one position counterclockwise from first element.
  - Rear gives position of last element.





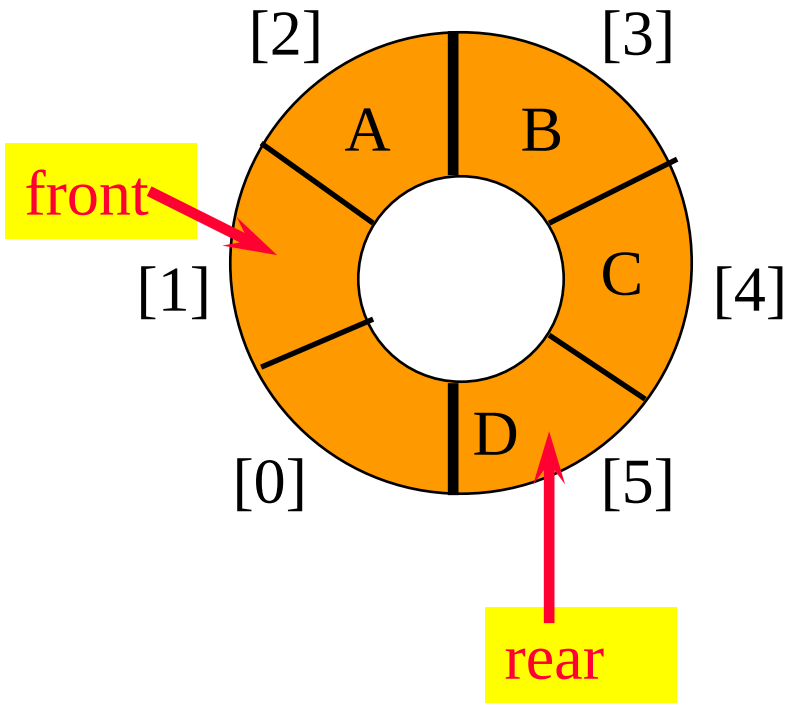
Push an Element

- Move rear one clockwise.



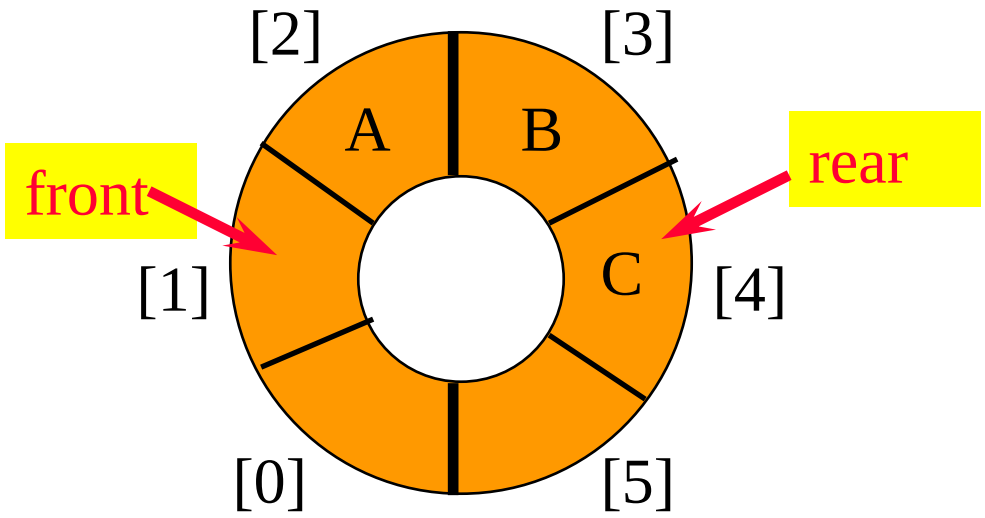
Push an Element

- Move rear one clockwise.
- Then put into queue[rear].



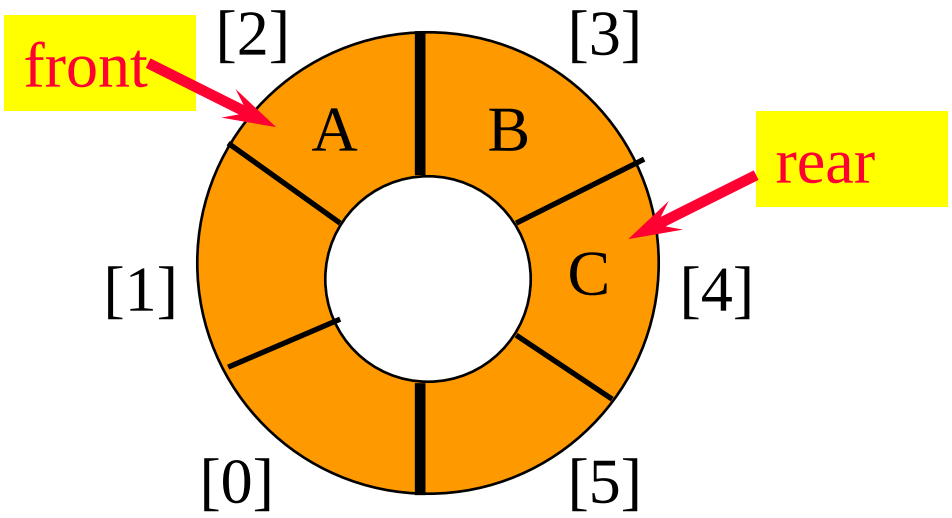
Pop an Element

- Move front one clockwise.



Pop an Element

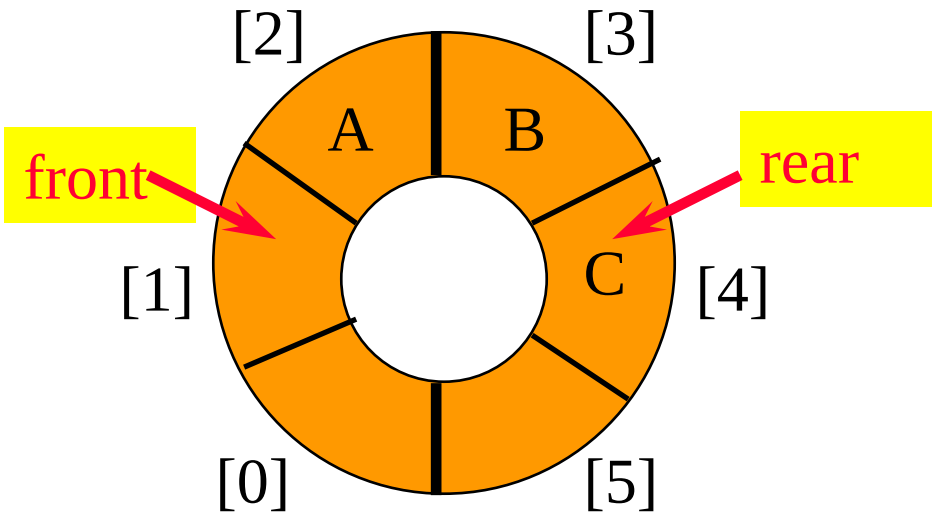
- Move front one clockwise.
- Then extract from queue[front].



Moving Rear Clockwise

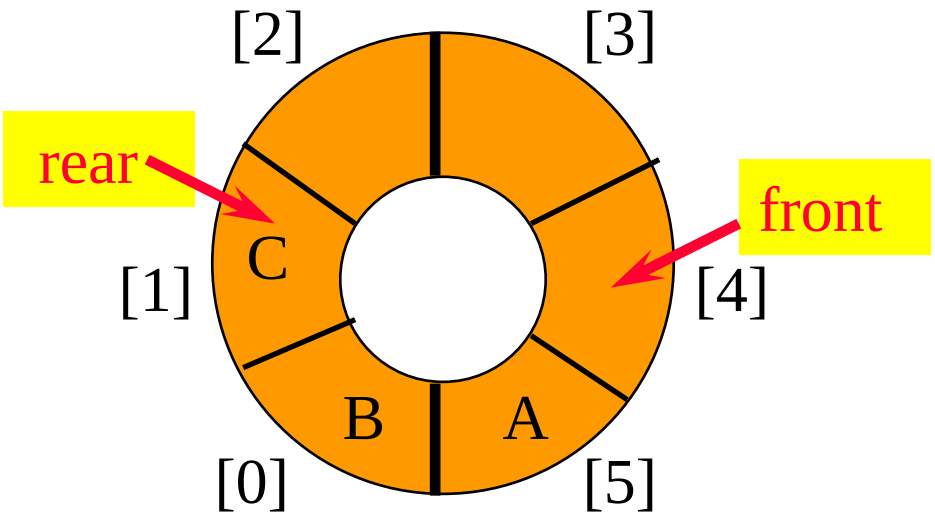
- rear++;

  if (rear = = capacity) rear = 0;

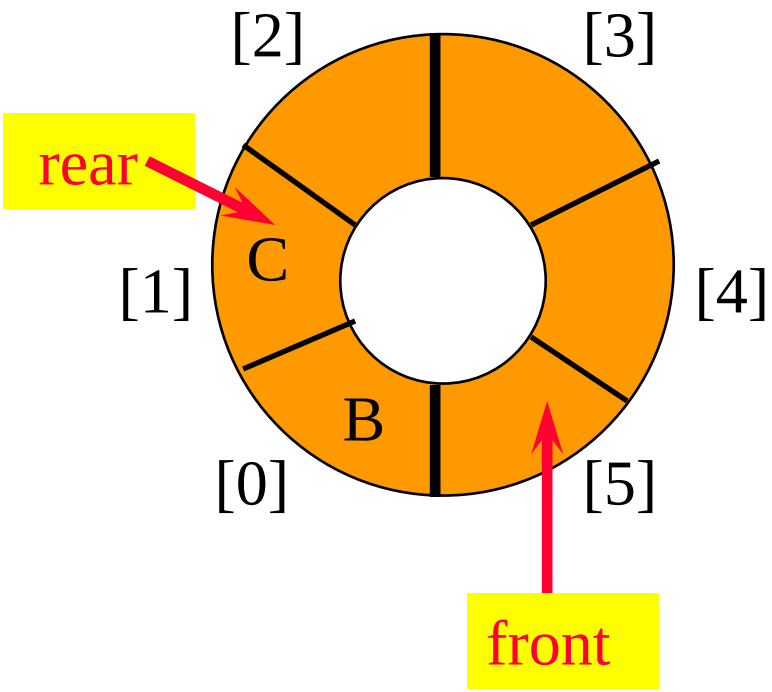


- rear = (rear + 1) % capacity;

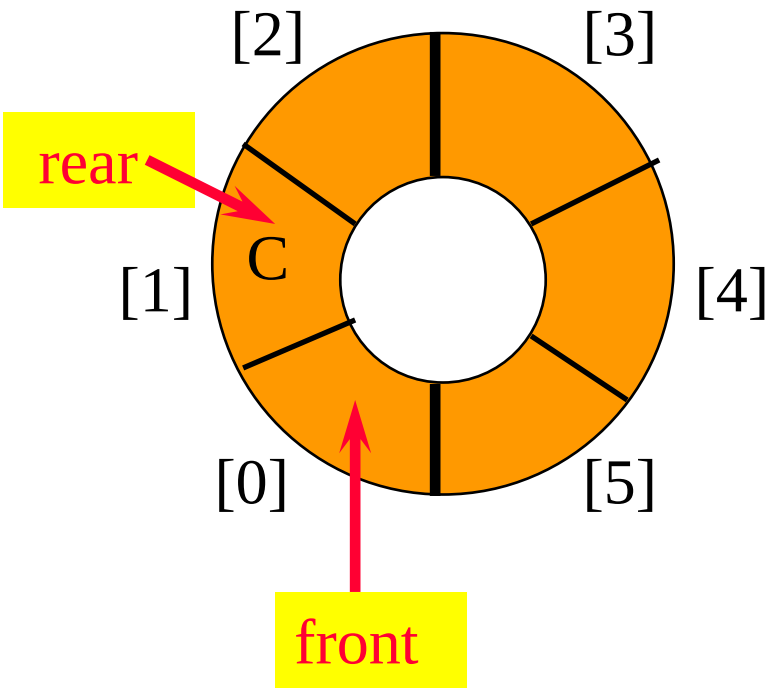
Empty That Queue



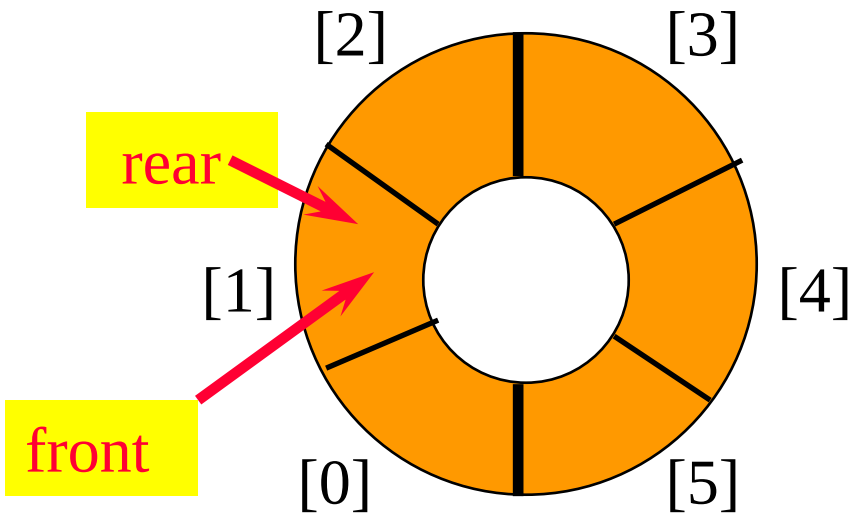
Empty That Queue



Empty That Queue

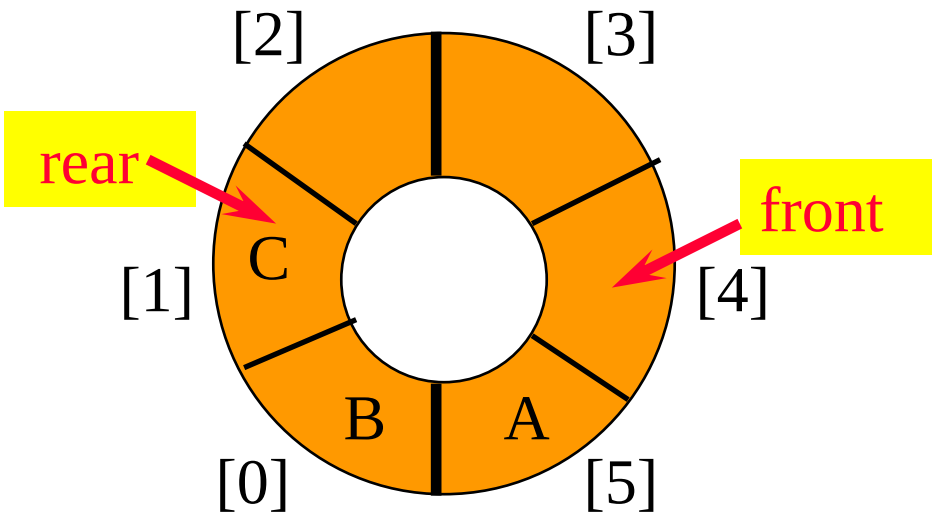


Empty That Queue

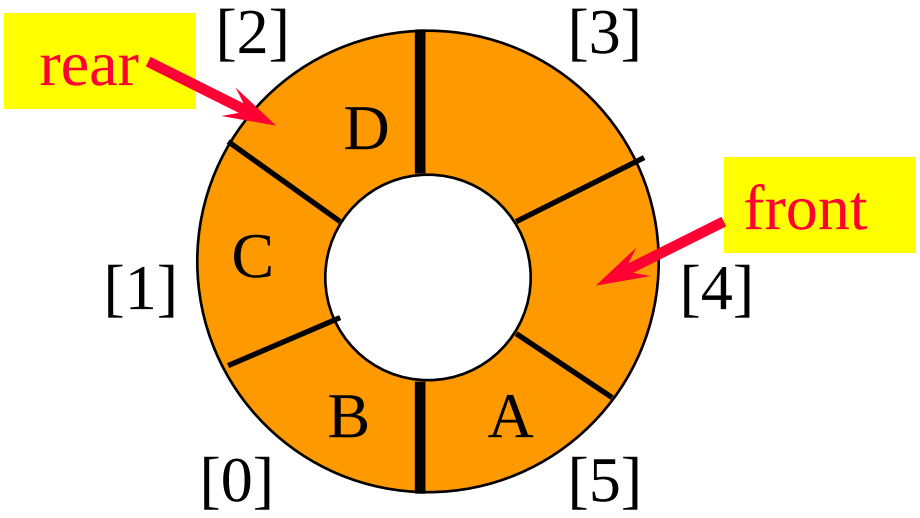


- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

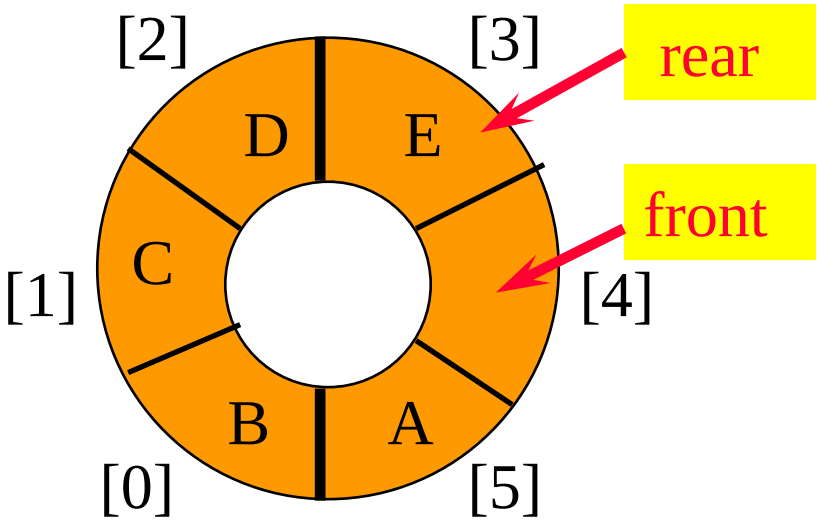
A Full Tank Please



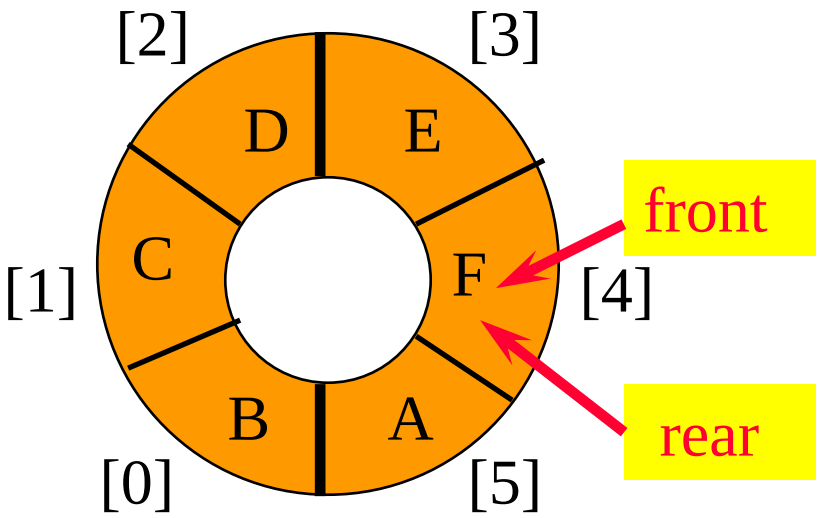
A Full Tank Please



A Full Tank Please



A Full Tank Please



- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!

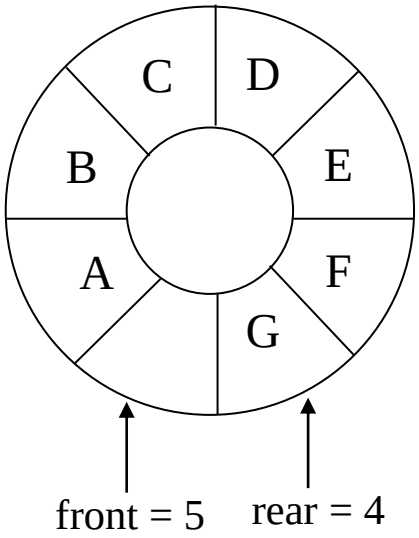
Ouch!!!!!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsPush.
    - Following each push set this variable to true.
    - Following each pop set to false.
    - Queue is empty iff (front == rear) && !lastOperationIsPush
    - Queue is full iff (front == rear) && lastOperationIsPush

Ouch!!!!!

- Remedies (continued).
  - Define an integer variable size.
    - Following each push do size++.
    - Following each pop do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == arrayLength)

Doubling Queue Capacity



(a) A full circular queue

queue [0] [1] [2] [3] [4] [5] [6] [7]

C	D	E	F	G		A	B

front = 5, rear = 4

(b) Flattened view of circular full queue

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10][11][12][13][14][15]

C	D	E	F	G		A	B								

front = 5, rear = 4

(c) After array doubling

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10][11][12][13][14][15]

C	D	E	F	G										A	B

front = 13, rear = 4

(d) After shifting right segment

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10][11][12][13][14][15]

A	B	C	D	E	F	G									

front = 15, rear = 6

(e) Alternative configuration

Linked Lists

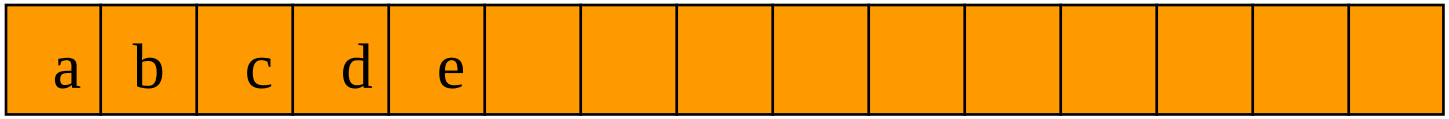
Linked Lists

- List elements are stored, in memory, in an arbitrary order.

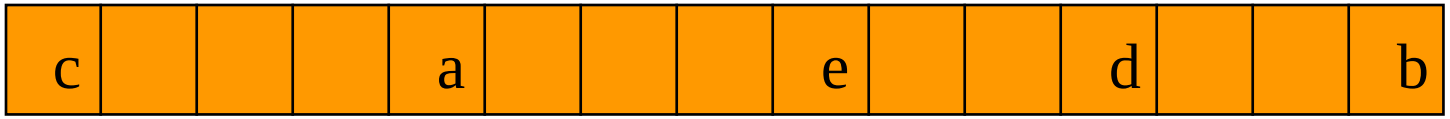
- Explicit information (called a link) is used to go from one element to the next.

Memory Layout

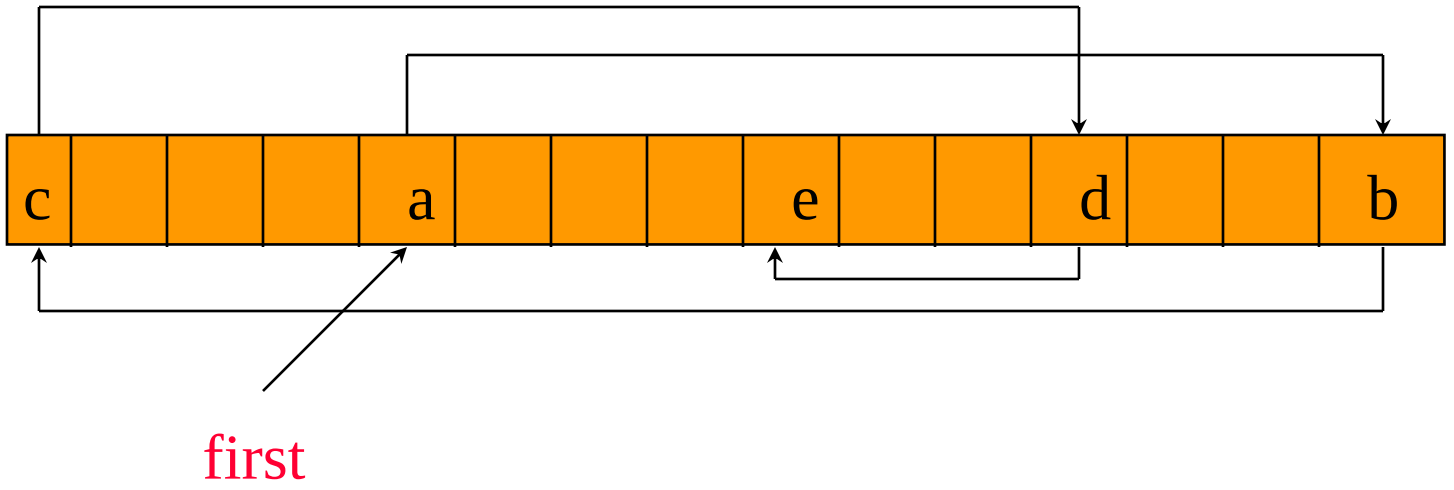
Layout of L = (a,b,c,d,e) using an array representation



A linked representation uses an arbitrary layout.



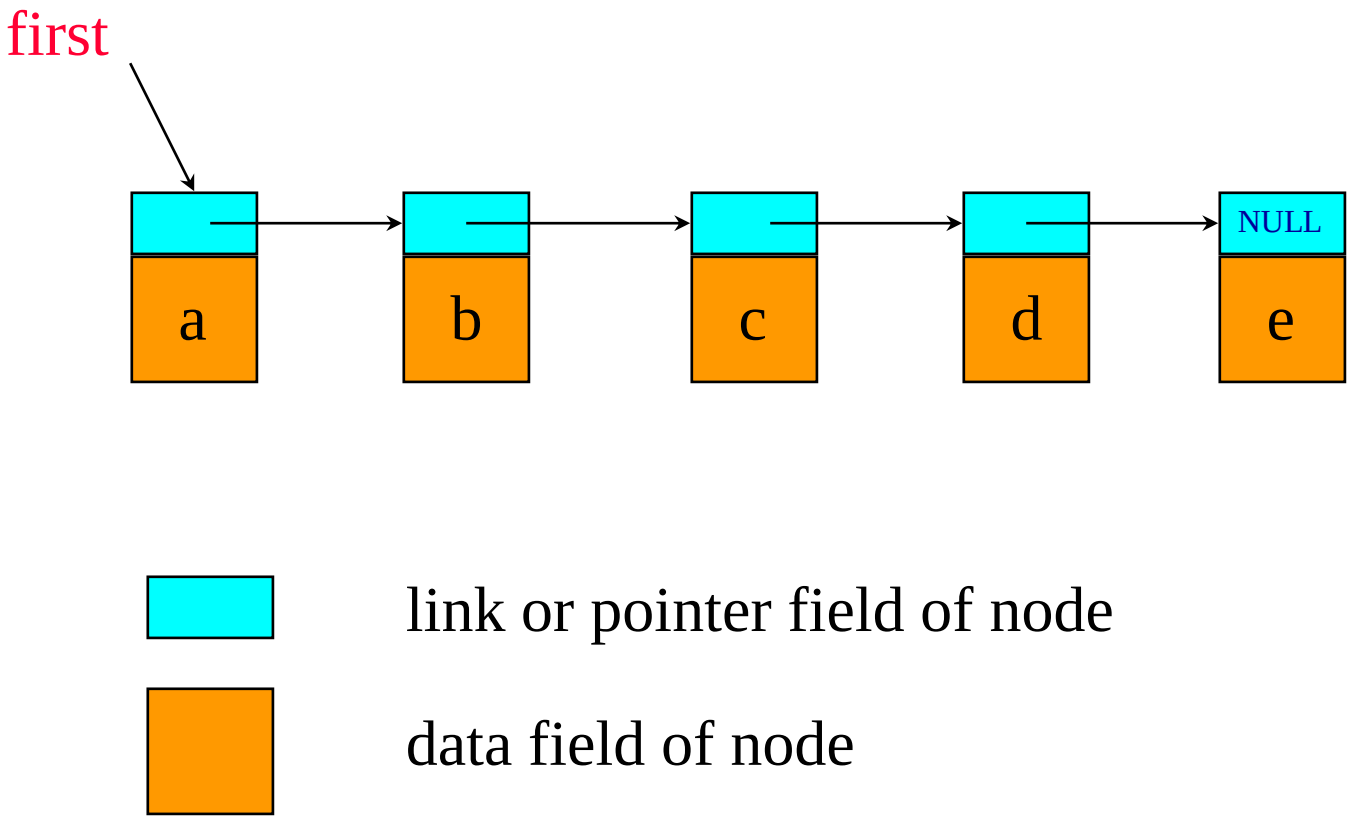
Linked Representation



Use a variable first to get to the first element a.

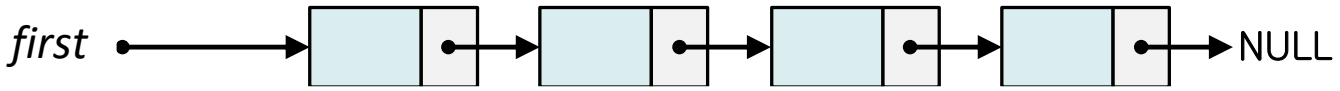
Pointer (or link) in e is NULL.

Normal Way to Draw a Linked List

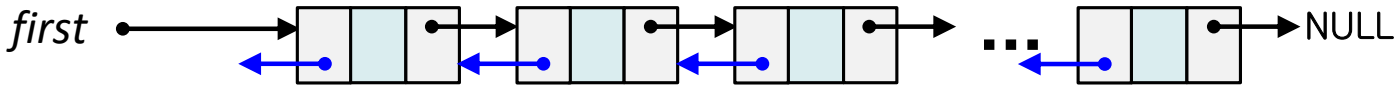


Types of Linked Lists

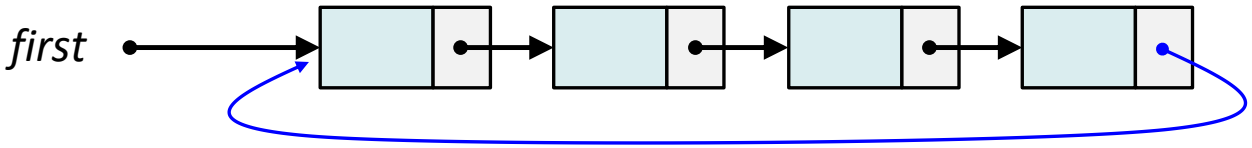
- Singly Linked List



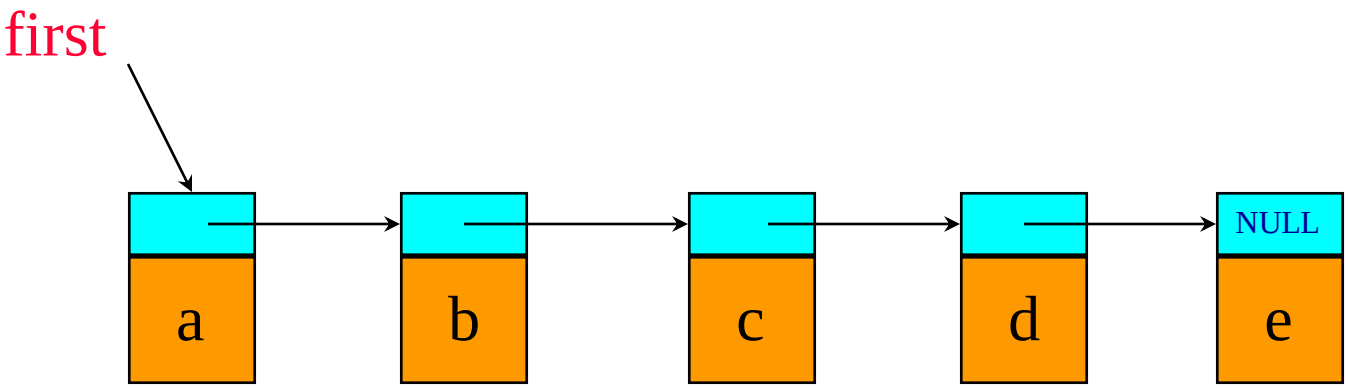
- Doubly Linked List



- (Singly/Doubly) Circular Linked List



Chain



- A chain is a singly linked list that is comprised of zero or more nodes.
- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer.

Node Representation

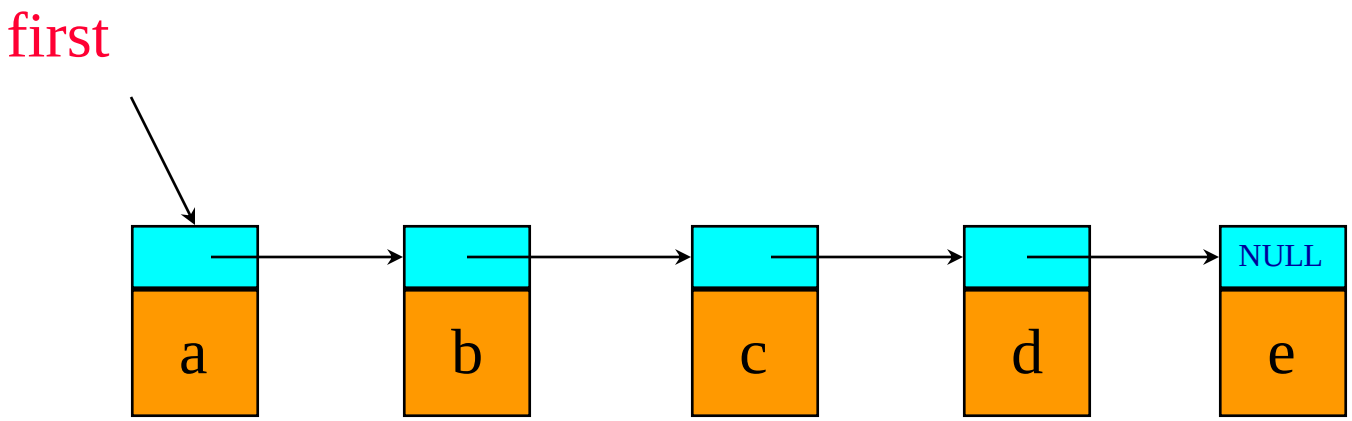
```
template <class T>
class Chain;

template <class T>
class ChainNode {
    template <class T> friend class Chain;

public:
    ChainNode(const T data, ChainNode<T>* link = NULL);

private:
    T data;
    ChainNode<T> *link;
};
```

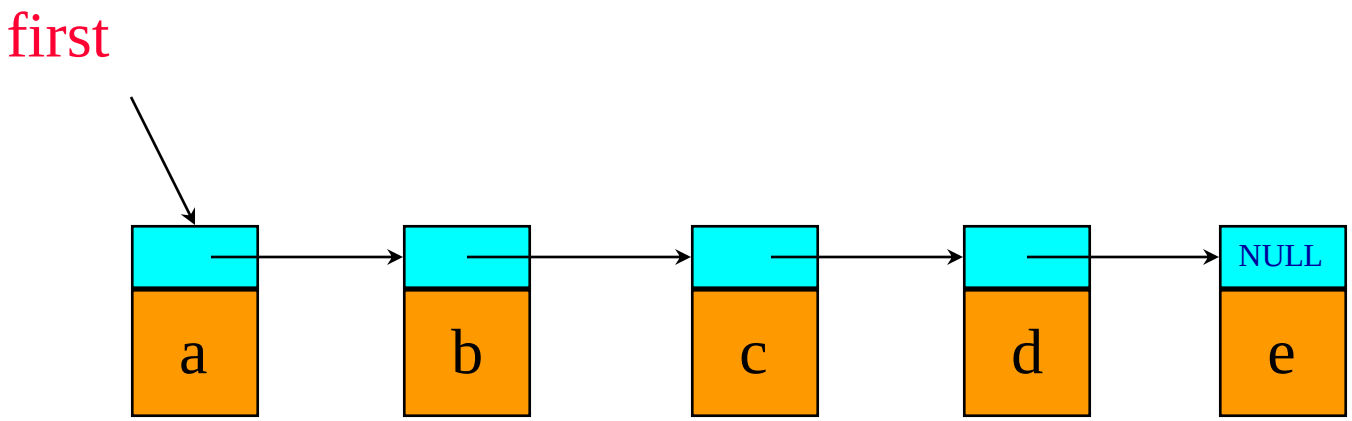
Get(0)



desiredNode = first; // gets you to first node

return desiredNode–>data;

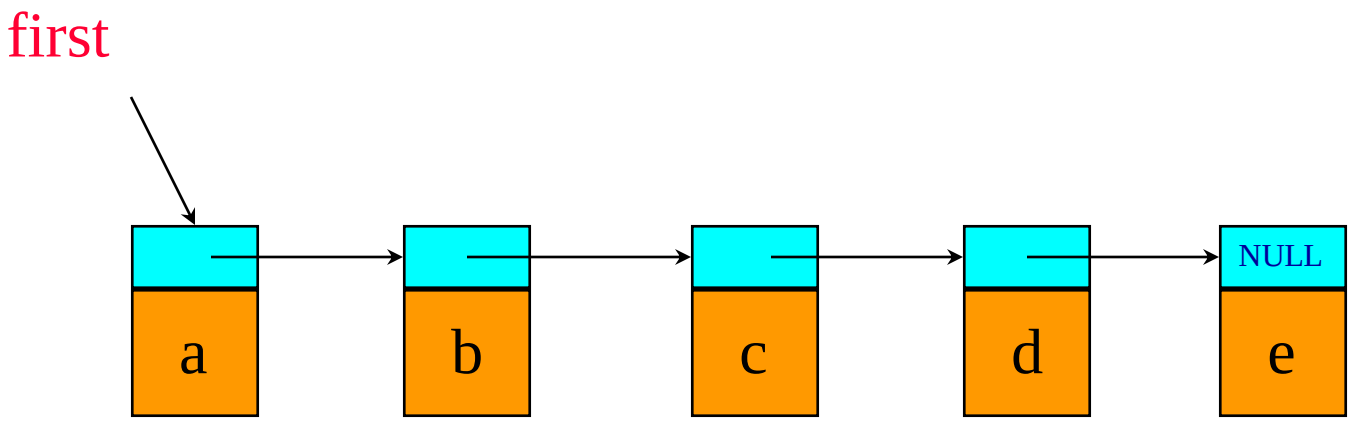
Get(1)



desiredNode = first–>link; // gets you to second node

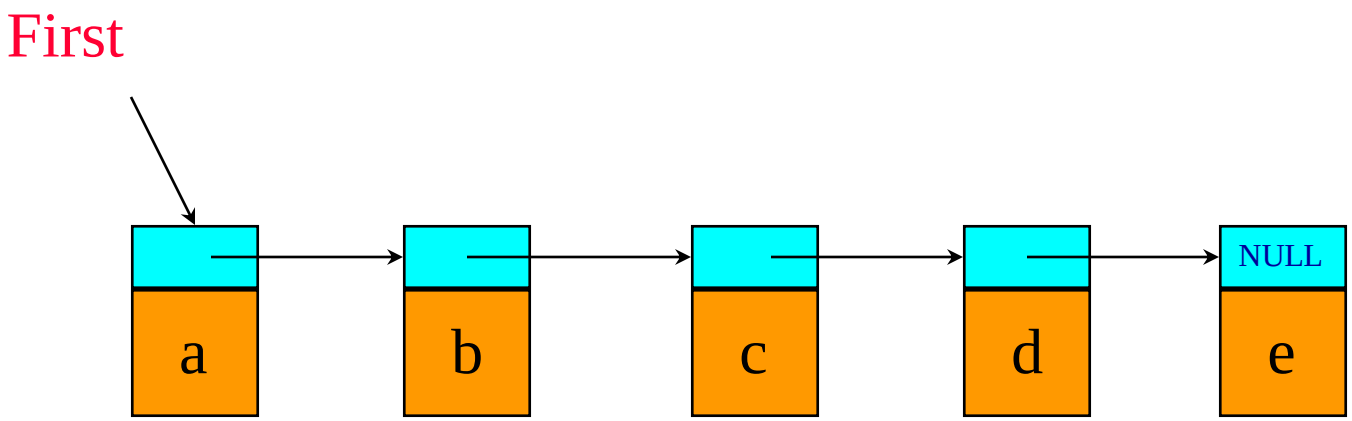
return desiredNode–>data;

Get(2)



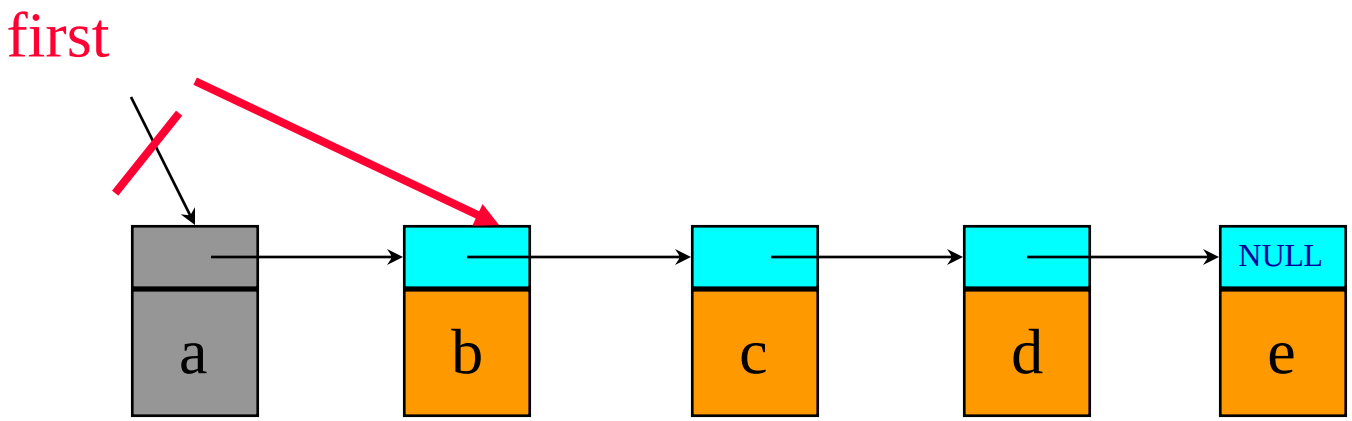

```
desiredNode = first–>link–>link; // gets you to third node
return desiredNode–>data;
```

Get(5)




```
desiredNode = first–>link–>link–>link–>link–>link;
                                    // desiredNode = NULL
return desiredNode–>data;   // NULL.element
```

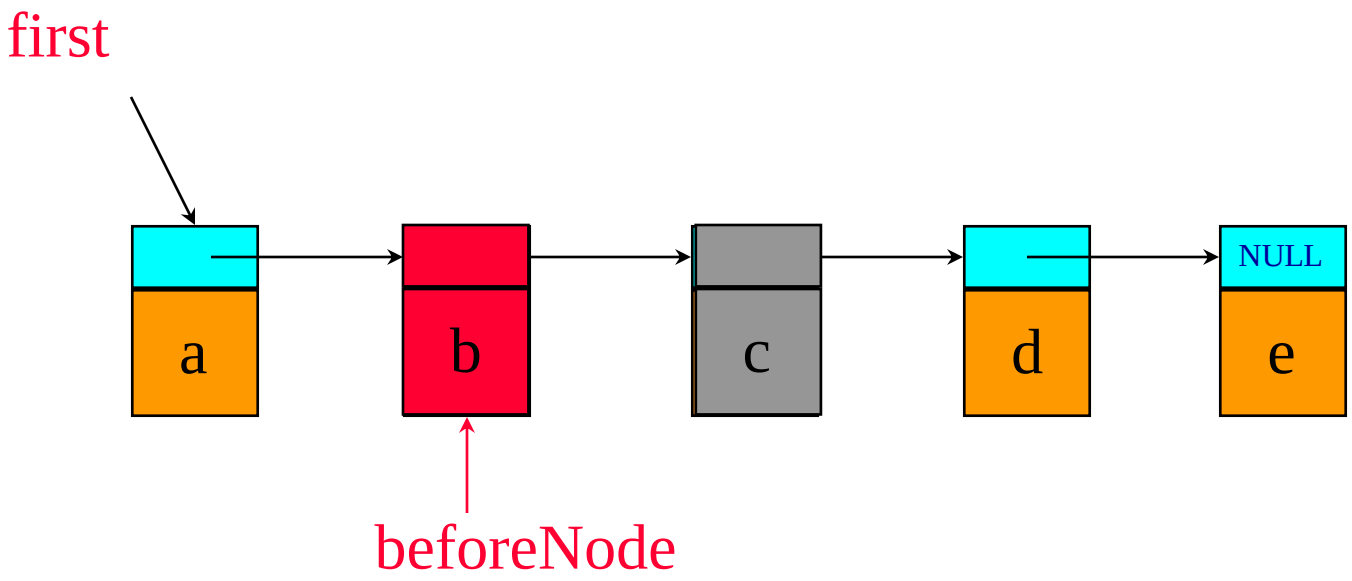
Delete an Element



Delete(0)

```
deleteNode = first;
first = first–>link;
delete deleteNode;
```

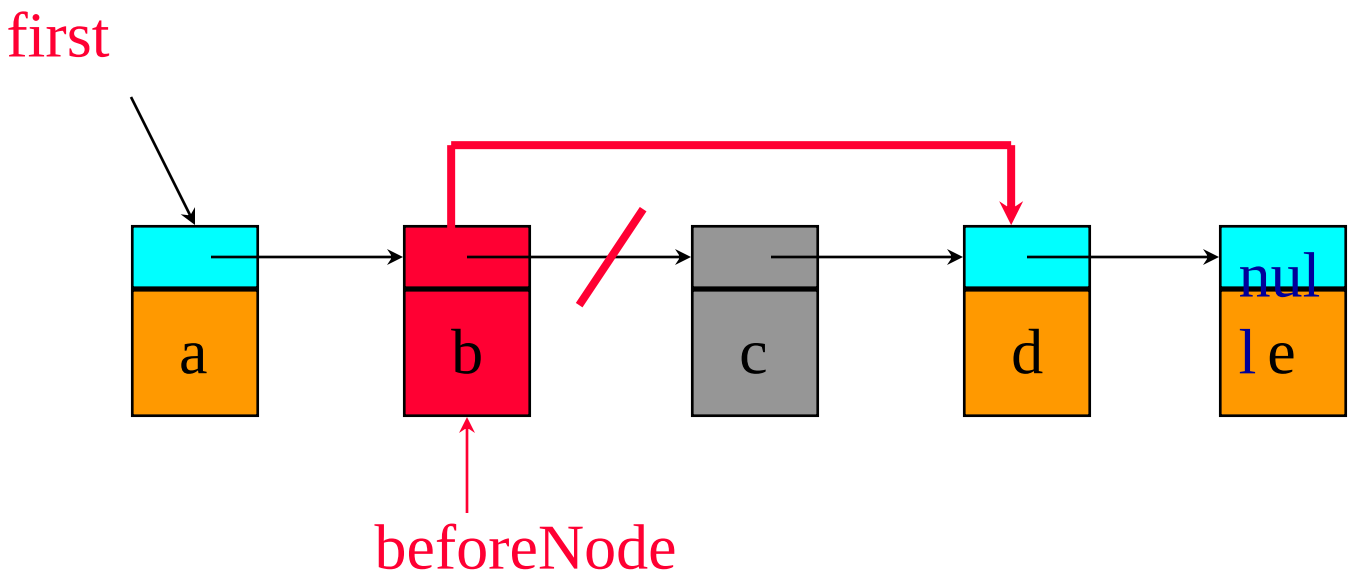
Delete(2)



first get to node just before node to be removed

```
beforeNode = first->link;
```

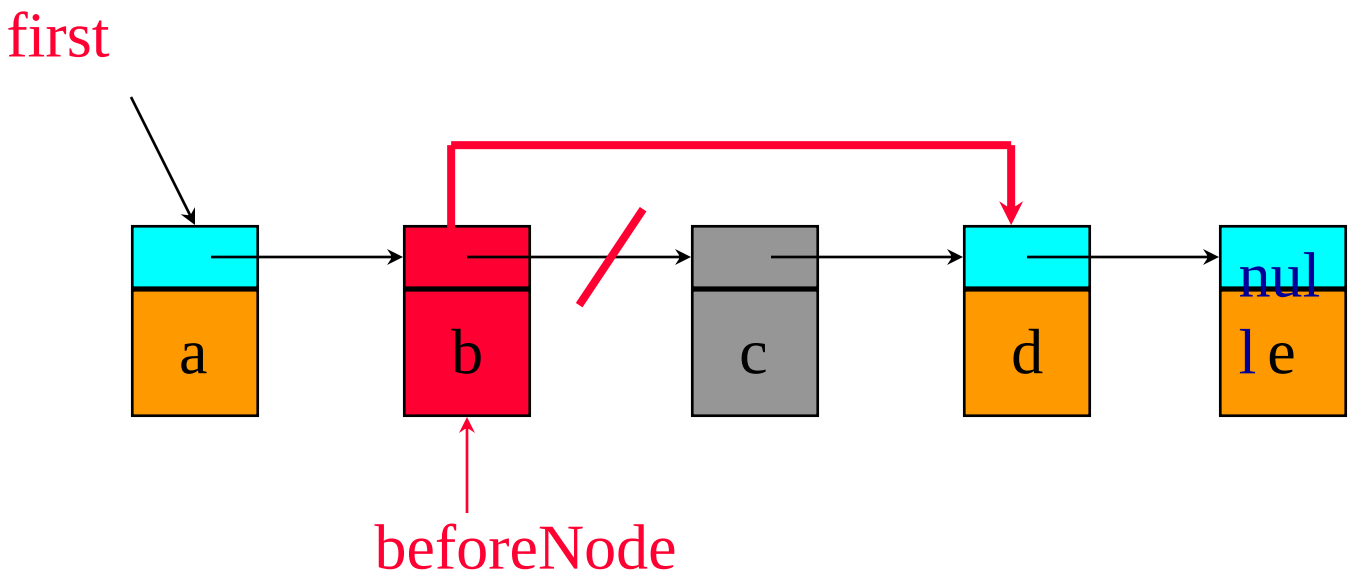
Delete(2)



save pointer to node that will be deleted

deleteNode = beforeNode–>link;

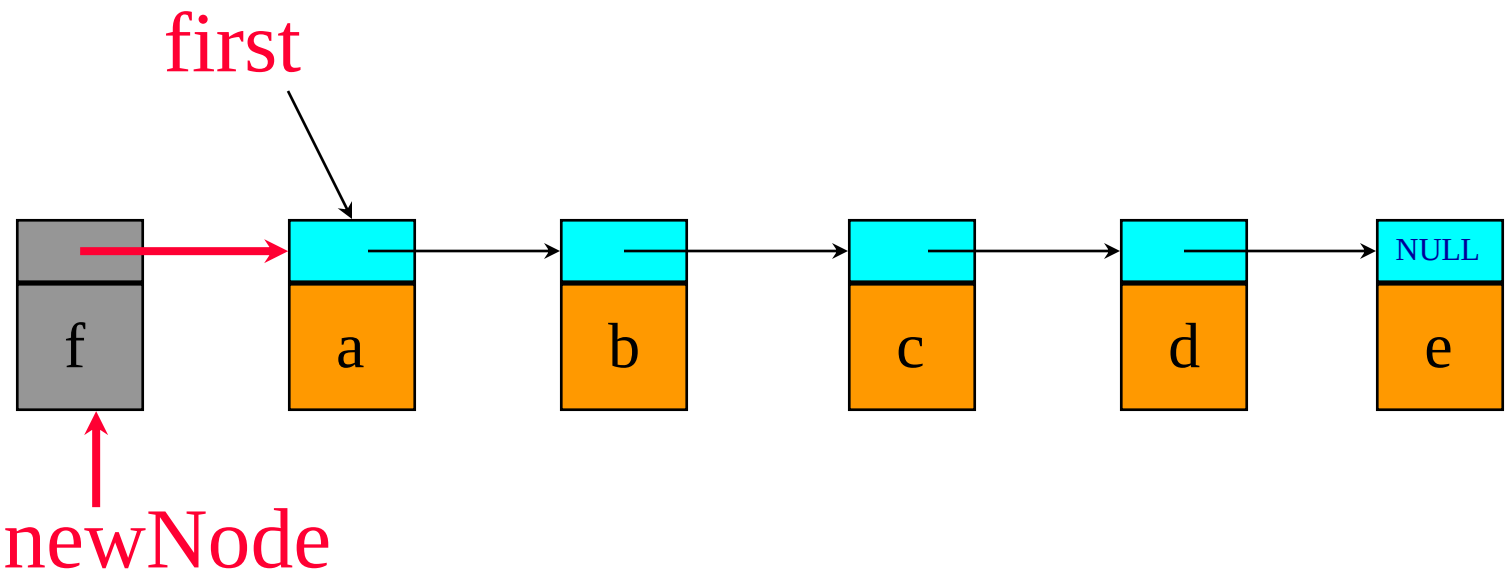
Delete(2)



now change pointer in beforeNode

```
beforeNode–>link = beforeNode–>link–>link;
delete deleteNode;
```

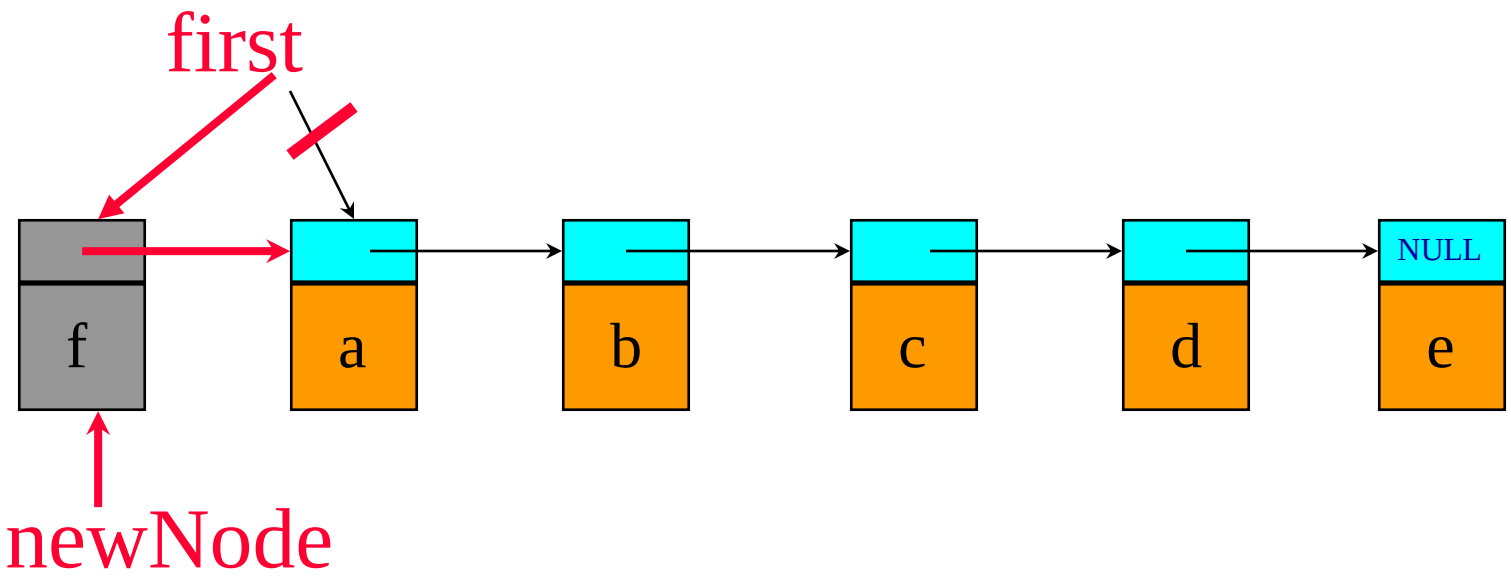
Insert(0,'f')



Step 1: get a node, set its data and link fields

```
newNode = new ChainNode<char>(theElement,
                              first);
```

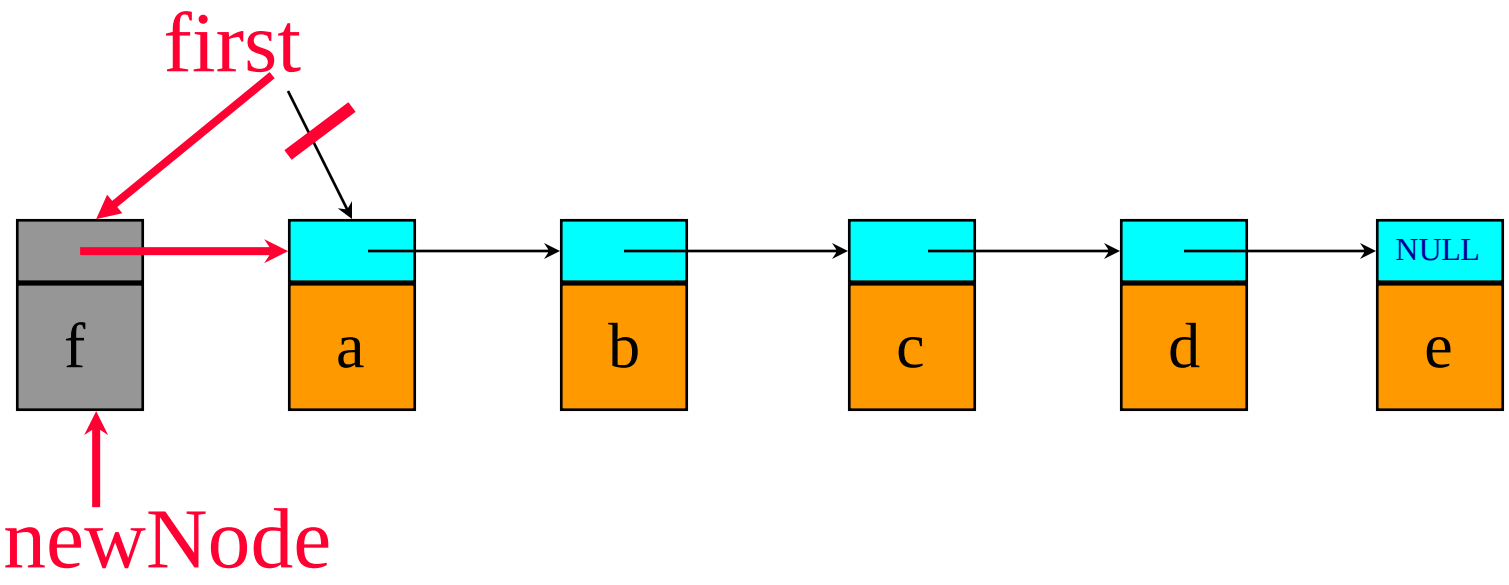
Insert(0,'f')



Step 2: update first

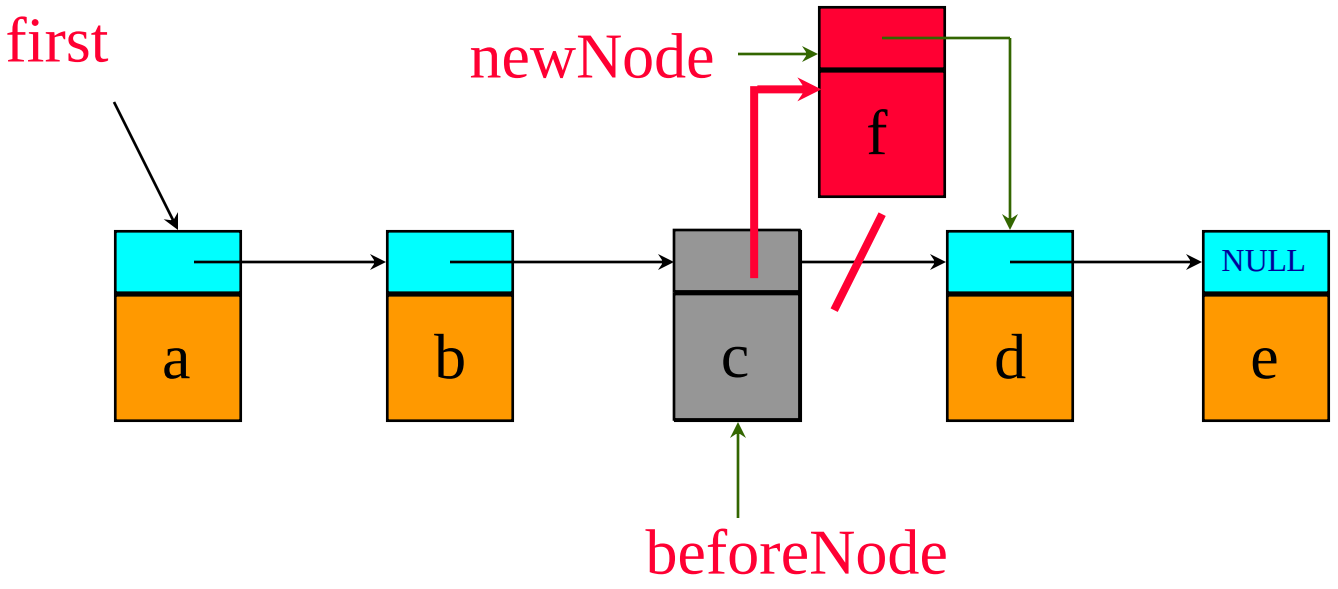
first = newNode;

One-Step Insert(0,'f')



first = new ChainNode<char>(‘f’, first);

Insert(3,'f')



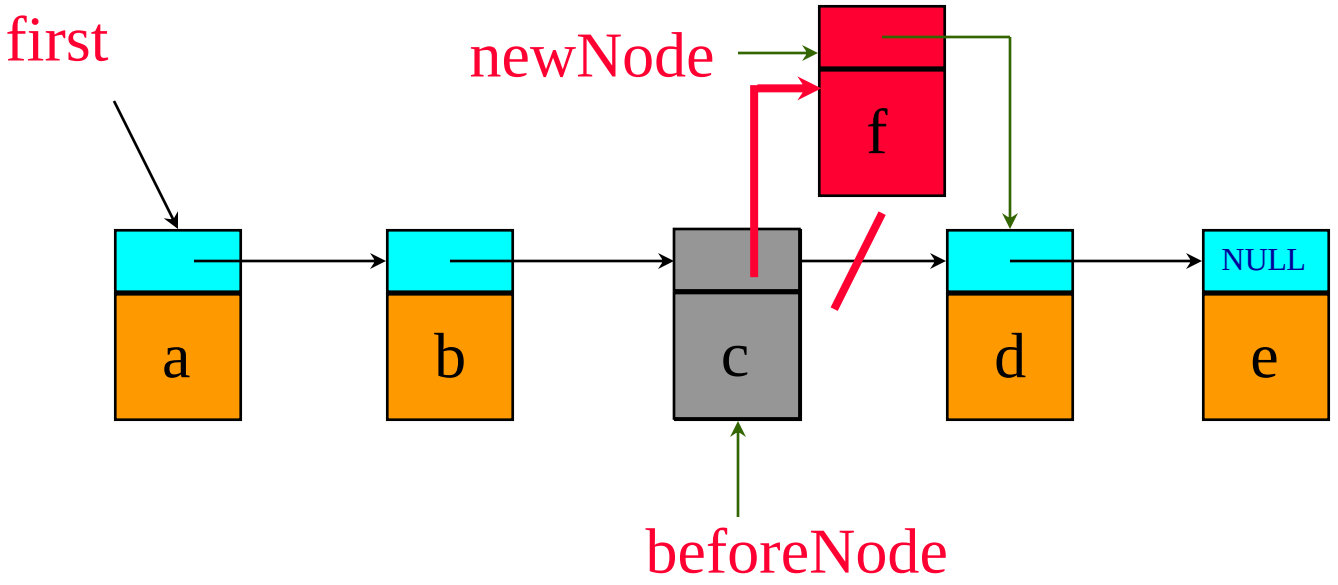
- first find node whose index is 2
- next create a node and set its data and link fields

```
ChainNode<char>* newNode = new ChainNode<char>( 'f',
                                                beforeNode->link);
```

- finally link beforeNode to newNode

```
beforeNode->link = newNode;
```

Two-Step Insert(3,'f')




```
beforeNode = first–>link–>link;
beforeNode–>link = new ChainNode<char>
                                ('f', beforeNode–>link);
```

Constructor of ChainNode

```
template <class T>
ChainNode<T>::ChainNode(const T data, ChainNode<T>* link) {
    this->data = data;
    this->link = link;
}
```

Chain Class

```
template <class T>
class Chain {
public:
    Chain(void);
    virtual ~Chain();
    bool IsEmpty(void);
    void StackPush(T data);
    void StackPop(void);
    void QueuePush(T data);
    void QueuePop(void);
    void Print(void);

private:
    ChainNode<T> *first;
    ChainNode<T> *last;
};
```

Virtual Destructor

- Here's when you need to make your destructor virtual:
  - if someone will derive from your class,
  - and if someone will say new Derived, where Derived is derived from your class,
  - and if someone will say delete p, where the actual object's type is Derived but the pointer p's type is your class.

Virtual Destructor

```
class Base{
public:
    ~Base() {
        cout << "Base destructor!" << endl;
    }
};

class Derived : public Base{
public:
    char* largeBuffer;
    Derived() {
        largeBuffer = new char[3000];
    }

    ~Derived() {
        cout << "Derived destructor!" << endl;
        delete[] largeBuffer;
    }
};
```

Virtual Destructor

```
int main(){
    //코드1
    cout << "---Derived* der1 = new Derived()---" << endl;
    Derived* der1 = new Derived();
    delete der1;

    //코드2
    cout << "\n\n---Base* der2 = new Derived()---" << endl;
    Base* der2 = new Derived();
    delete der2;

}
```

```
---Derived* der1 = new Derived()---
Derived destructor!
Base destructor!


---Base* der2 = new Derived()---
Base destructor!
```

Constructor & Destructor of Chain

```
template <class T>
Chain<T>::Chain(void) {
    first = NULL;
    last = NULL;
}

template <class T>
Chain<T>::~Chain() {
    ChainNode<T> *next = NULL;

    while (first != NULL)
    {
        next = first->link;
        cout << "Delete: " << first->data << endl;
        delete first;
        first = next;
    }
}
```

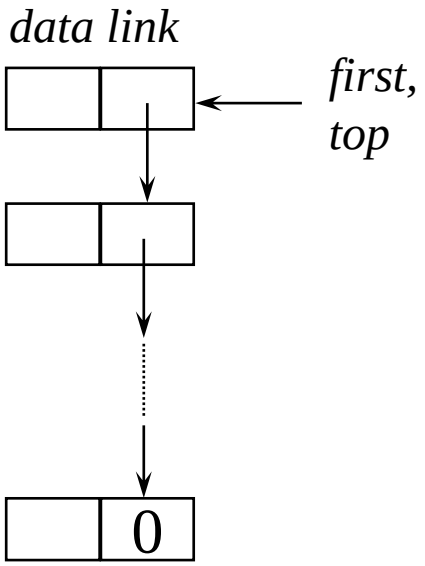
IsEmpty & Print of Chain

```
template <class T>
bool Chain<T>::IsEmpty(void) {
    return first == NULL;
}

template <class T>
void Chain<T>::Print(void)
{
    ChainNode<T> *current = first;

    cout << "Print: ";
    while (current != NULL) {
        cout << current->data << ' ';
        current = current->link;
    }
    cout << endl;
}
```

Linked Stack

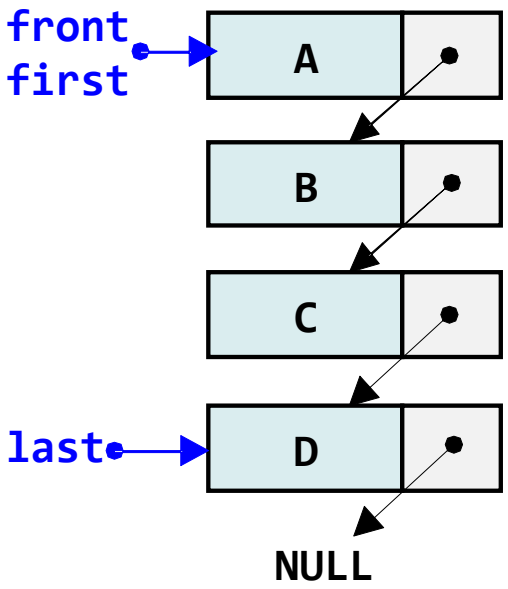



```
template <class T>
void Chain<T>::StackPush(T data) {
    first = new ChainNode<T>(data, first);
    cout << "StackPush: " << data << endl;
}

template <class T>
void Chain<T>::StackPop(void) {
    ChainNode<T> *top = first;

    if(IsEmpty()) {
        cout << "StackPop: empty!" << endl;
    }
    else {
        cout << "StackPop: " << top->data << endl;
        first = top->link;
        delete top;
    }
}
```

Linked Queue




```
template <class T>
void Chain<T>::QueuePush(T data) {
    if (IsEmpty())
        first = last = new ChainNode<T>(data, NULL);
    else
        last = last->link = new ChainNode<T>(data, NULL);
    cout << "QueuePush: " << data << endl;
}

template <class T>
void Chain<T>::QueuePop(void) {
    ChainNode<T> *front = first;

    if (IsEmpty()) {
        cout << "QueuePop: empty!" << endl;
    }
    else {
        cout << "QueuePop: " << front->data << endl;
        first = front->link;
        delete front;
    }
}
```

main

```
int main(void)
{
    Chain<int> chain;

    cout << "--- Test: Linked Stack" << endl;
    chain.StackPush(1);
    chain.StackPush(2);
    chain.StackPush(3);

    chain.Print();

    chain.StackPop();
    chain.StackPop();
    chain.StackPop();
    chain.StackPop();

    cout << endl << "--- Test: Linked Queue" << endl;
    chain.QueuePush(1);
    chain.QueuePush(2);
    chain.QueuePush(3);

    chain.Print();

    chain.QueuePop();
    chain.QueuePop();
    chain.QueuePop();
    chain.QueuePop();
}
```

Homework #1

- Implement Circular Queue.
- Implement Stack and Queue by inheriting the Chain class.
- Homework을 제출할 필요는 없으나
  중간/기말고사에 출제할 계획임