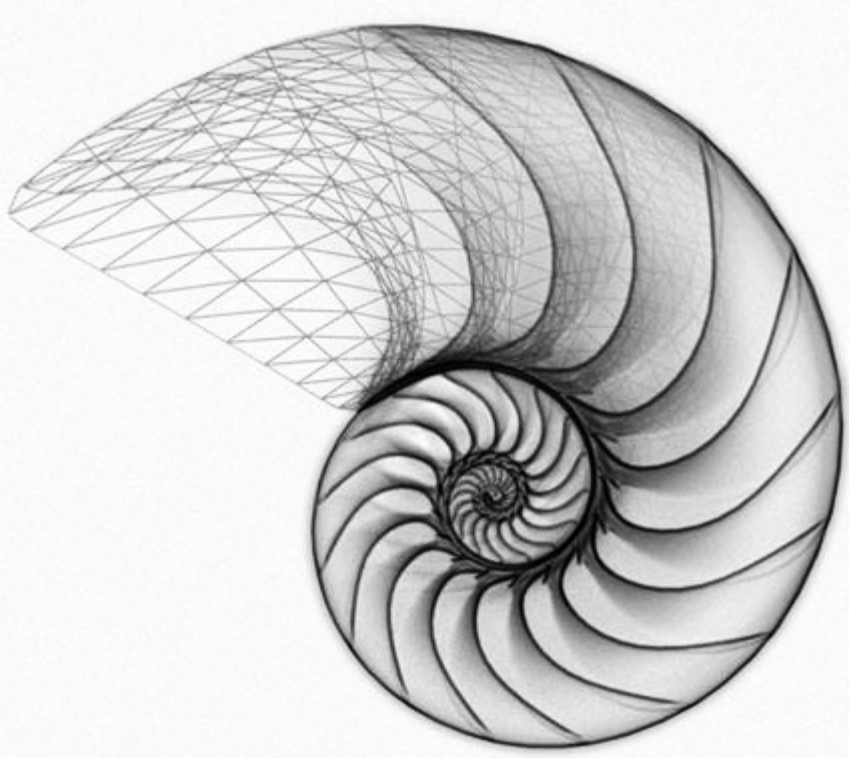


Lecture 13

Abstract Data Type Stack and Queue (Linked-list-based Implementation)

CSE225: Data Structures and Algorithms

Stack

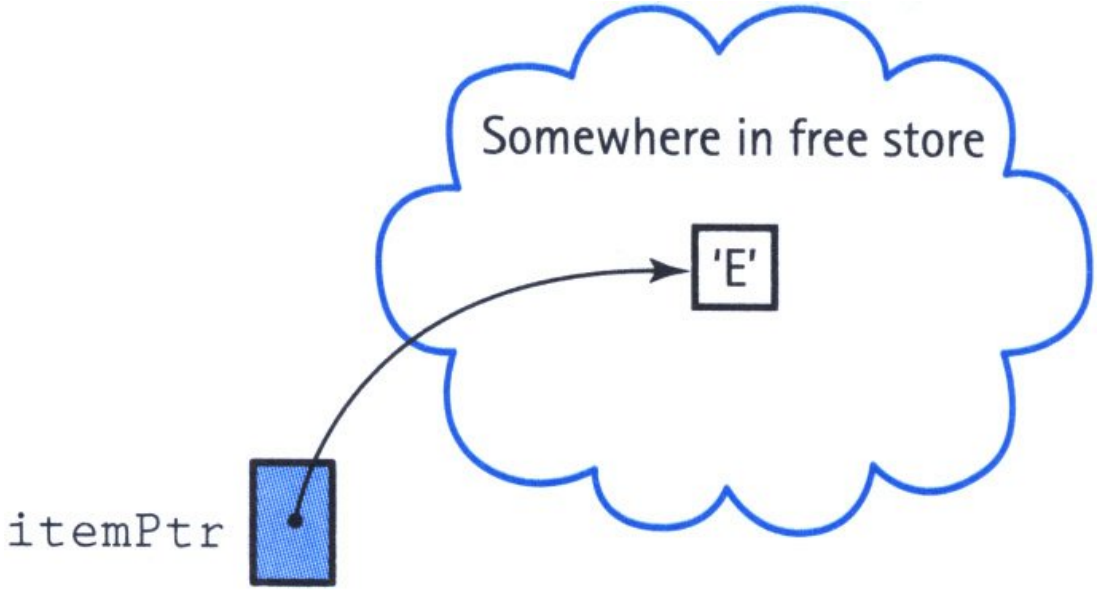
- Array based implementation
  - Static / dynamic array can be used
  - In either case, you **need to anticipate the size of stack before you use it**

- What if you could have a stack of unlimited size (or at least the maximum your RAM can support)?

Stack

- The elements in the stack now require to be dynamically allocated

```
// Allocate space for new item.
char *itemPtr = new char;
*itemPtr = 'E';
```

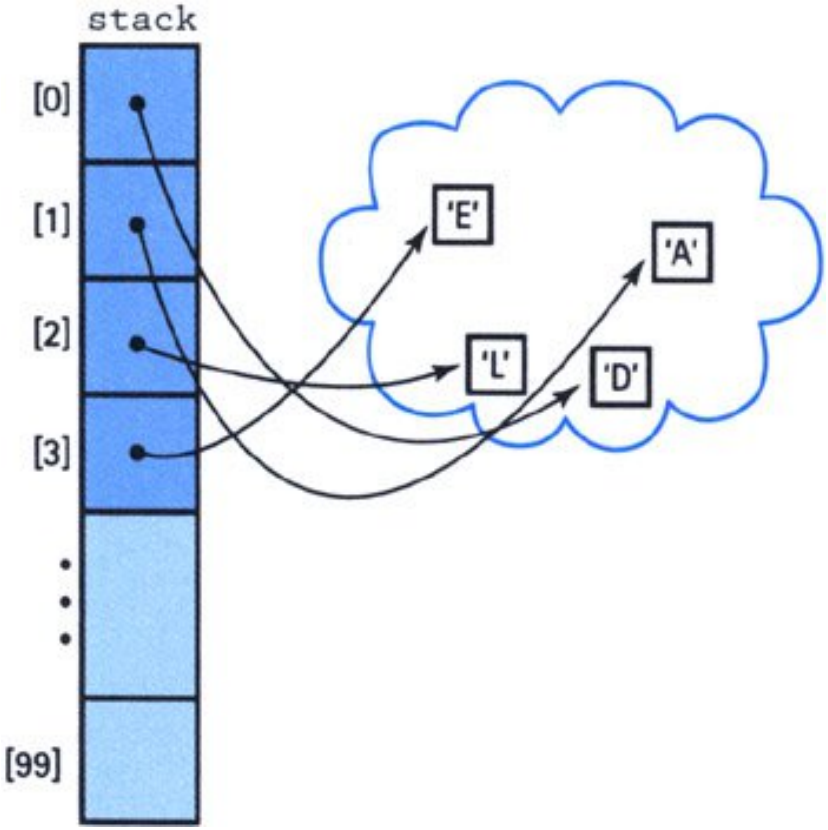


Stack

- But we are going to have multiple elements in the stack
- One pointer for each of the elements allocated dynamically
- How do we store these pointers?

Stack

- An array for all these pointers???

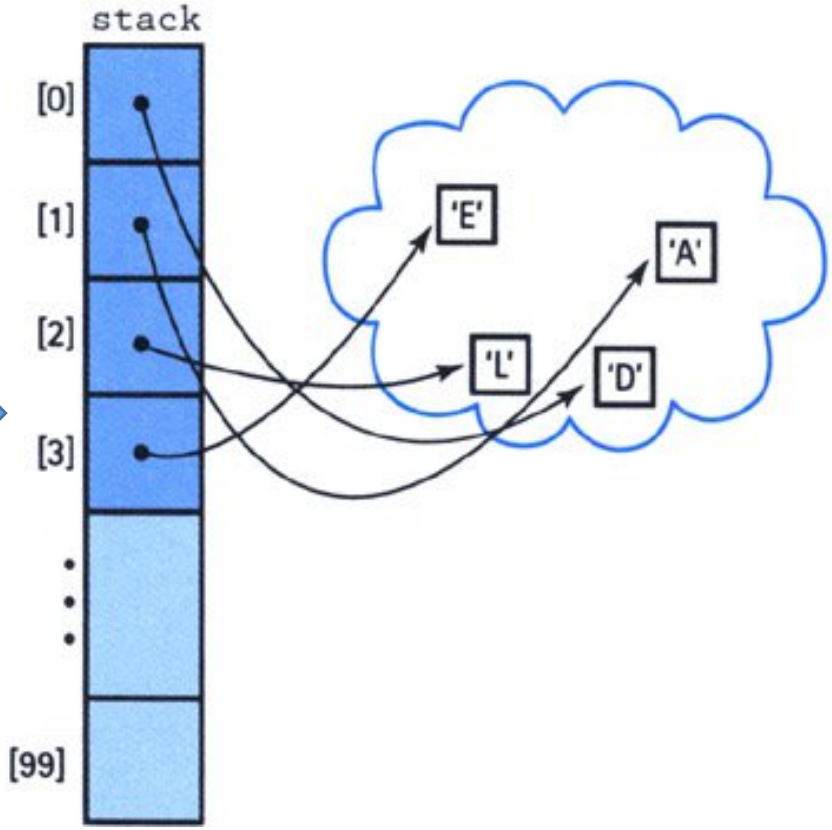


Stack

- An array for all these pointers???

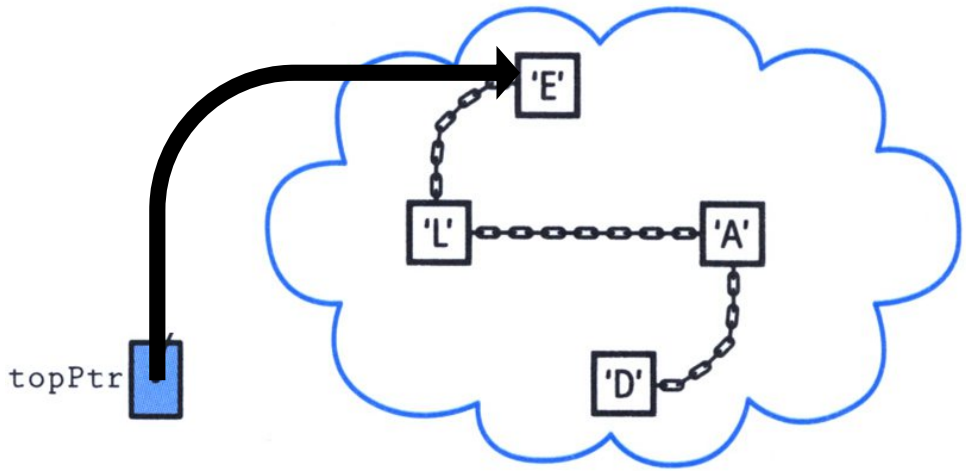
- This array can be dynamically allocated too

- Still we need to know the size of this array beforehand



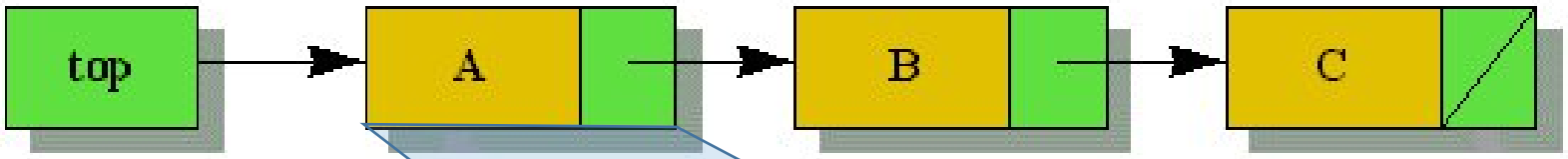
Stack

- A better solution is chaining the elements one after another

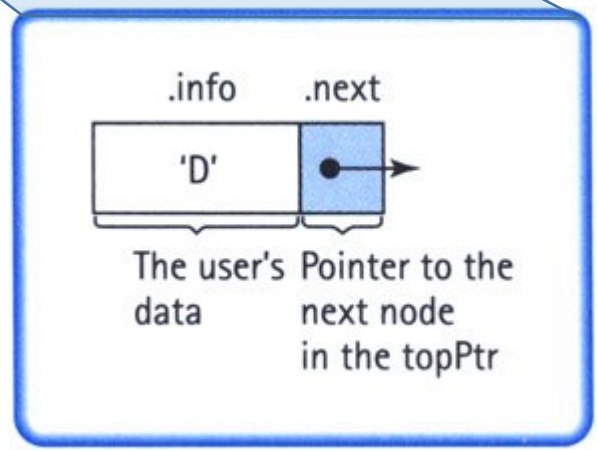


We do not need to know the size of the stack anymore. All we need is the top pointer.

Stack Implemented as Linked-List

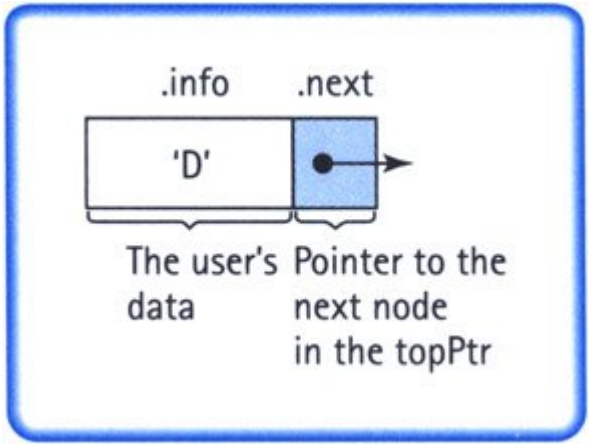



```
template <class ItemType>
struct NodeType
{
  ItemType info;
  NodeType* next;
};
```



Stack Implemented as Linked-List

```
template <class ItemType>
struct NodeType
{
  ItemType info;
  NodeType* next;
};
```



<u>2 ways of representing a node:</u>

```
NodeType<char> node, *nodePtr;
nodePtr = new NodeType<char>;
node.info = 'D';
nodePtr->info = 'L';
```

stacktype.h

```
#ifndef STACKTYPE_H_INCLUDED
#define STACKTYPE_H_INCLUDED

class FullStack
// Exception class used by Push
when stack is full.
{};

class EmptyStack
// Exception class used by Pop
and Top when stack is empty.
{};
```

```
template <class ItemType>
class StackType
{
  struct NodeType
  {
    ItemType info;
    NodeType* next;
  };
public:
  StackType();
  ~StackType();
  void Push(ItemType);
  void Pop();
  ItemType Top();
  bool IsEmpty();
  // bool IsFull();
private:
  NodeType<ItemType>* topPtr;
};

#endif // STACKTYPE_H_INCLUDED
```

stacktype.cpp

```
template <class ItemType>
StackType<ItemType>::StackType()
{
  topPtr = NULL;
}

template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
  return (topPtr == NULL);
}

template <class ItemType>
ItemType StackType<ItemType>::Top()
{
  if (IsEmpty())
    throw EmptyStack();
  else
    return topPtr->info;
}
```

stacktype.cpp

```
template <class ItemType>
StackType<ItemType>::StackType()
{
  topPtr = NULL;
}
```

O(1)

```
template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
  return (topPtr == NULL);
}
```

O(1)

```
template <class ItemType>
ItemType StackType<ItemType>::Top()
{
  if (IsEmpty())
    throw EmptyStack();
  else
    return topPtr->info;
}
```

O(1)

stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
}
```

stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

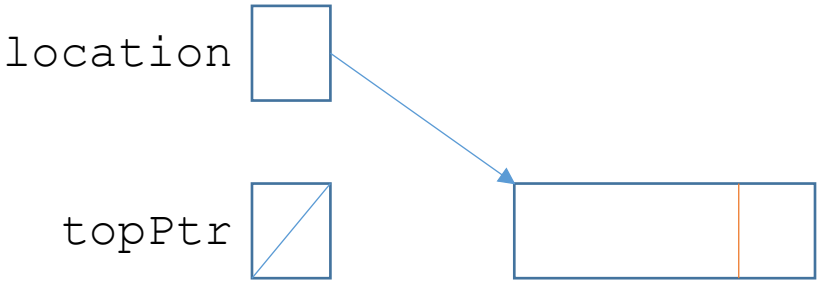
push('A');

topPtr

stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

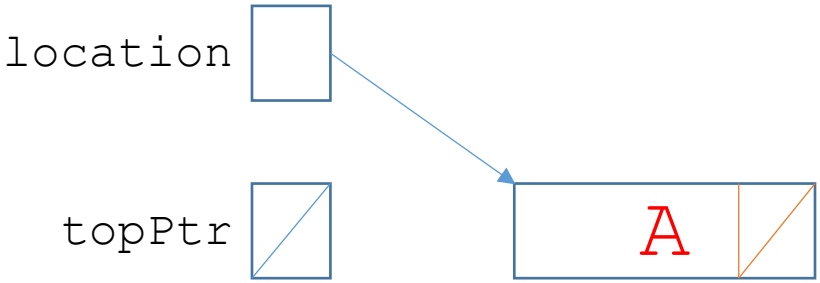
push('A');



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

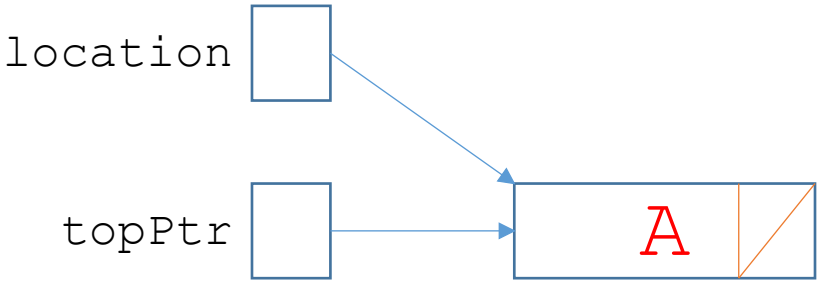
push('A');



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

push('A');



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

stacktype.cpp

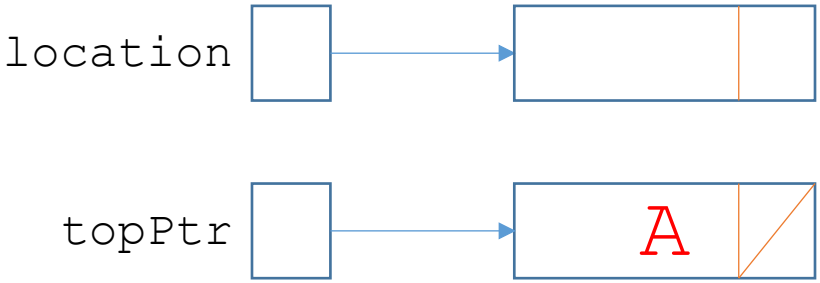
```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

push('B');

stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

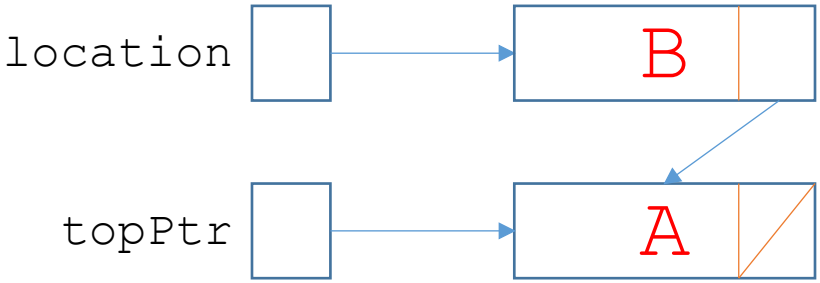
push('B');



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

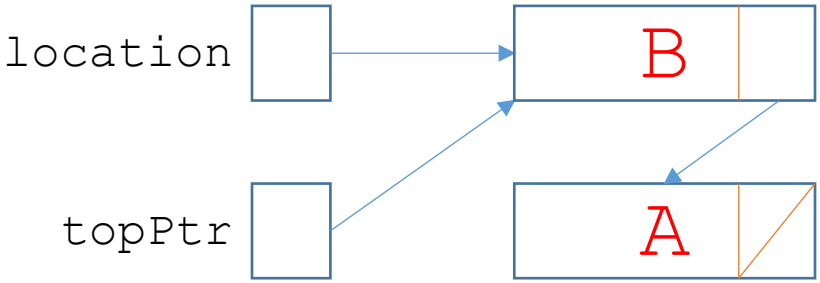
push('B');



stacktype.cpp

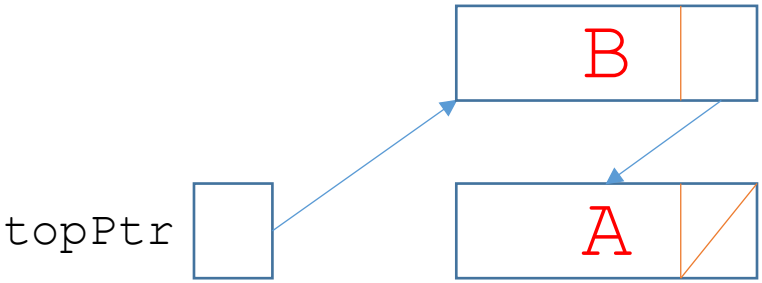
```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

push('B');



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
  if (IsFull())
    throw FullStack();
  else
  {
    NodeType* location;
    location = new NodeType;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
  }
}
```

O(1)

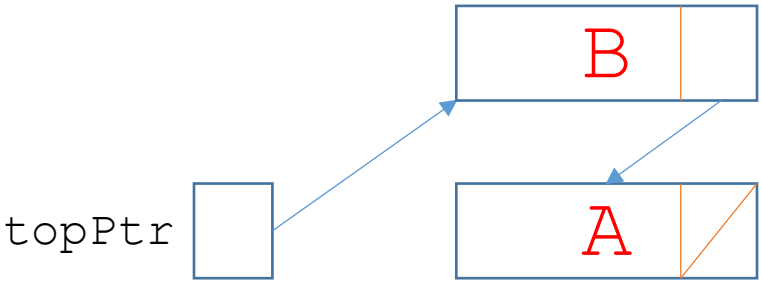
stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

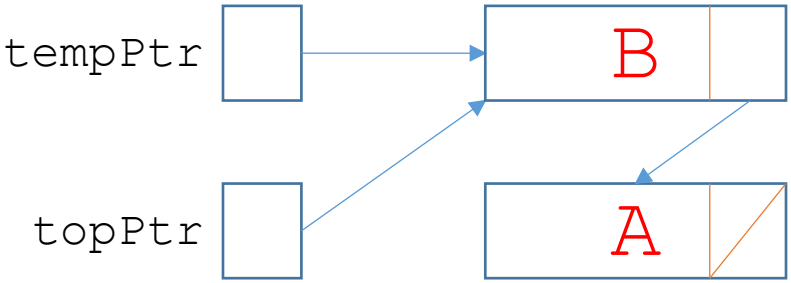
pop();



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

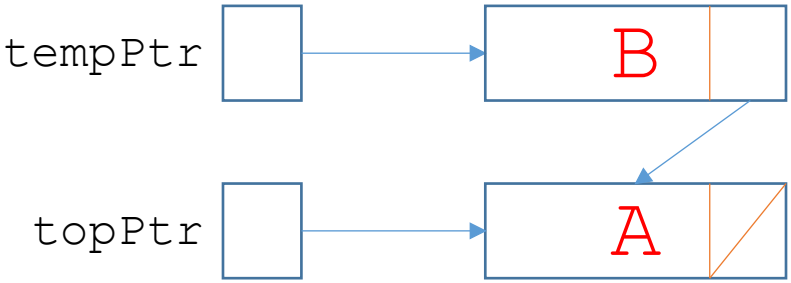
pop();



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

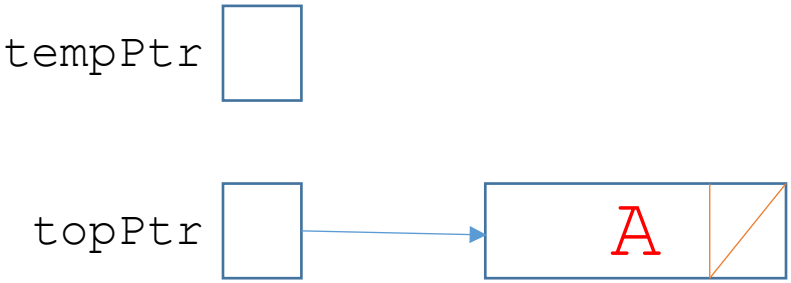
pop();



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

pop();



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

stacktype.cpp

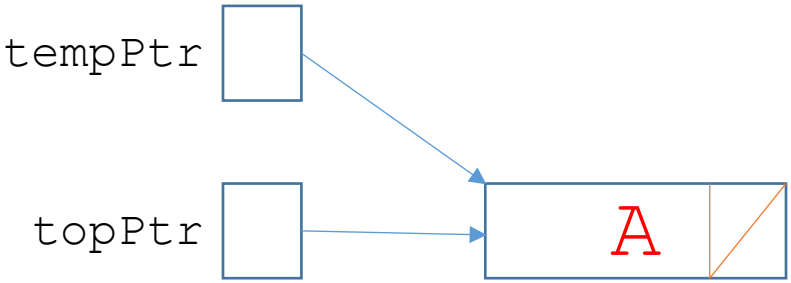
```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

pop();

stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

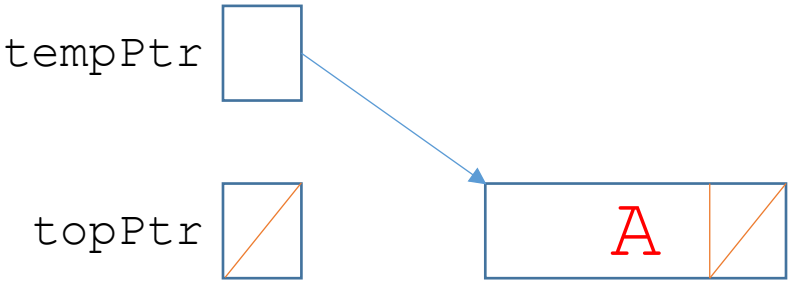
pop();



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

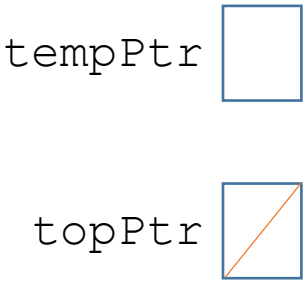
pop();



stacktype.cpp

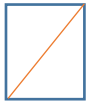
```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

pop();



stacktype.cpp

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

topPtr 

stacktype.cpp

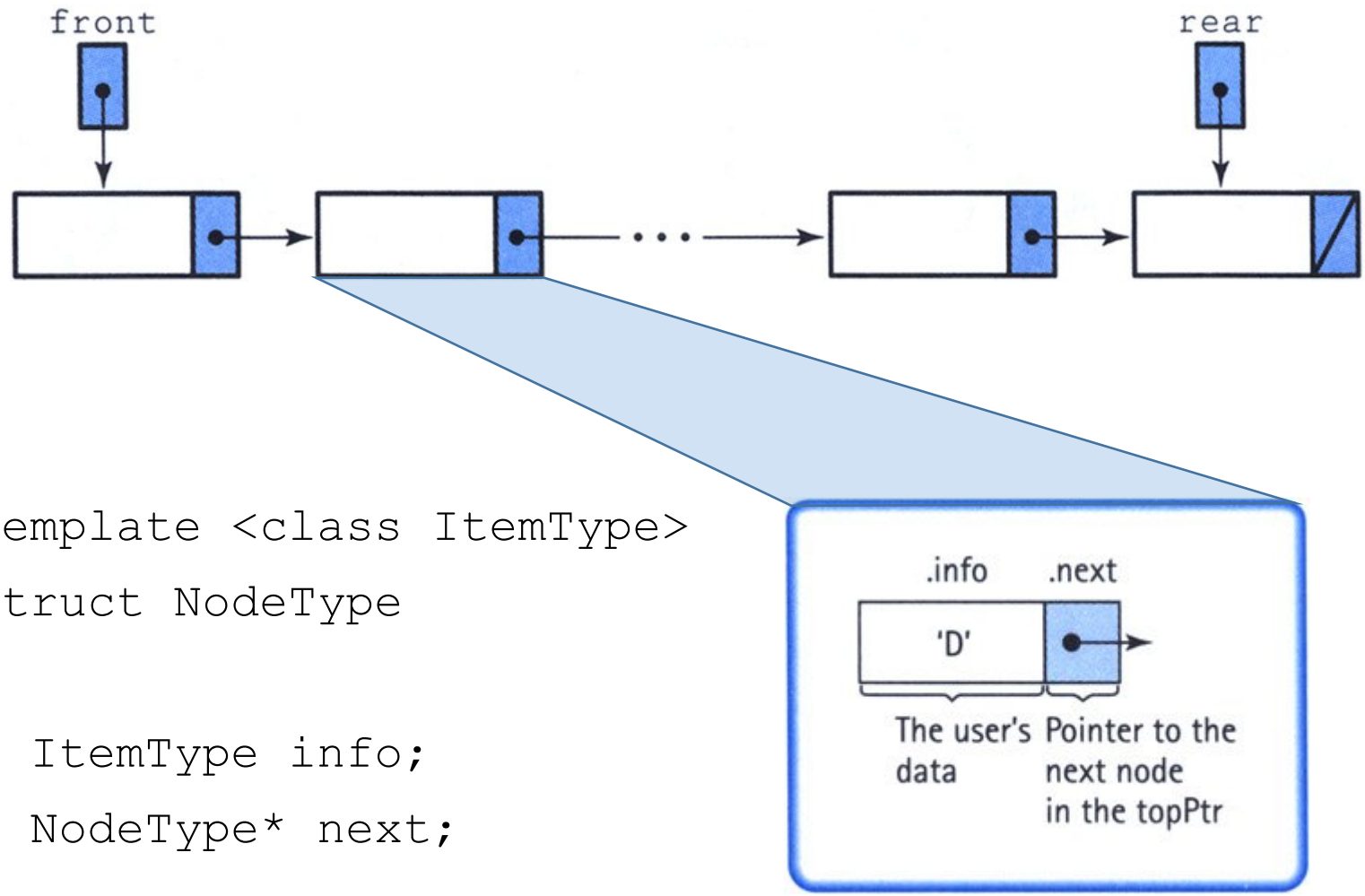
```
template <class ItemType>
void StackType<ItemType>::Pop()
{
  if (IsEmpty())
    throw EmptyStack();
  else
  {
    NodeType* tempPtr;
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}
```

O(1)

Queue

- Array based implementation
  - Just like stack, you need to anticipate the size of queue beforehand
  - No way to change its size during program execution
- Linked list based implementation
  - The queue grows and shrinks on demand

Queue Implemented as Linked-List




```
template <class ItemType>
struct NodeType
{
  ItemType info;
  NodeType* next;
};
```

queuetype.h

```
#ifndef QUEUETYPE_H_INCLUDED
#define QUEUETYPE_H_INCLUDED

class FullQueue
// Exception class used by Push
when queue is full.
{};

class EmptyQueue
// Exception class used by Pop
and Top when queue is empty.
{};
```

```
template <class ItemType>
class QueType
{
  struct NodeType
  {
    ItemType info;
    NodeType* next;
  };
public:
  QueType();
  ~QueType();
  void MakeEmpty();
  void Enqueue(ItemType);
  void Dequeue(&ItemType&);
  bool IsEmpty();
  //bool IsFull();
private:
  NodeType<ItemType>* front;
  NodeType<ItemType>* rear;
};

#endif // QUEUETYPE_H_INCLUDED
```

queuetype.cpp

```
template <class ItemType>
QueType<ItemType>::QueType()
{
  front = NULL;
  rear = NULL;
}

template <class ItemType>
bool QueType<ItemType>::IsEmpty()
{
  return (front == NULL);
}
```

```
template<class ItemType>
bool QueType<ItemType>::IsFull()
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc& exception)
  {
    return true;
  }
}
```

queuetype.cpp

```
template <class ItemType>
QueType<ItemType>::QueType()
{
  front = NULL;
  rear = NULL;
}
```

O(1)

```
template <class ItemType>
bool QueType<ItemType>::IsEmpty()
{
  return (front == NULL);
}
```

O(1)

```
template<class ItemType>
bool QueType<ItemType>::IsFull()
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc& exception)
  {
    return true;
  }
}
```

O(1)

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```

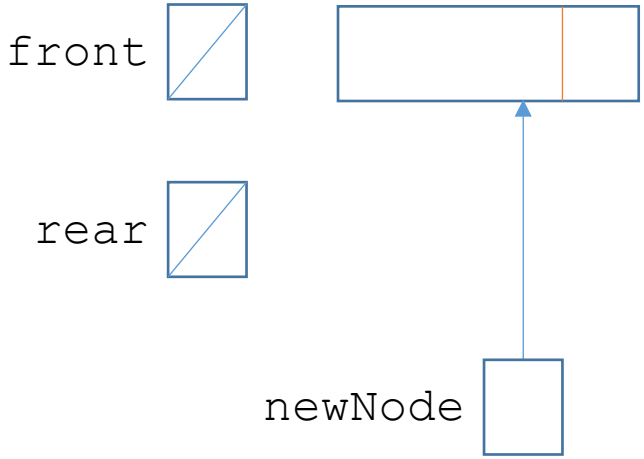
front

rear

Enqueue('A')

queuetype.cpp

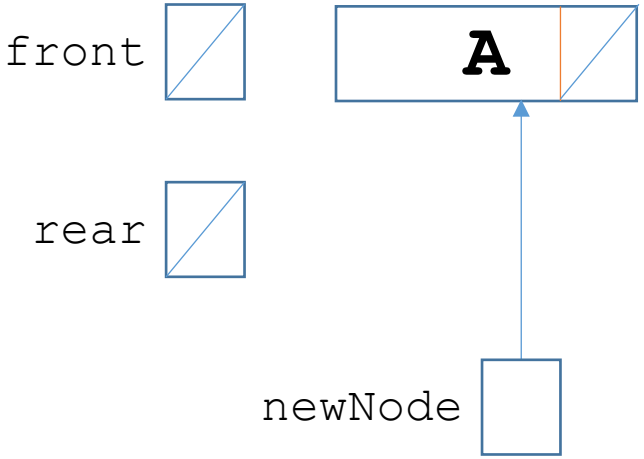
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('A')

queuetype.cpp

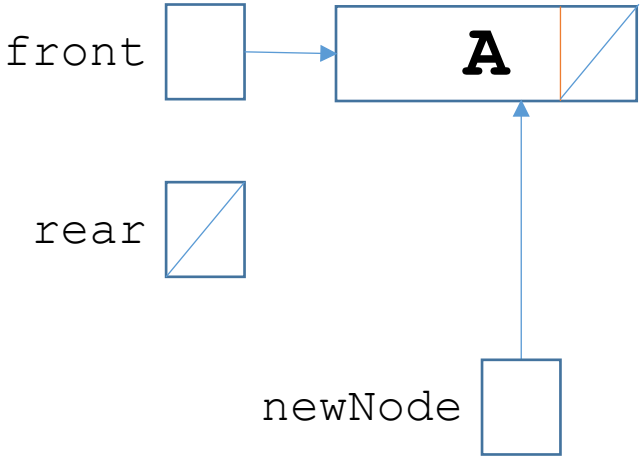
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;


   rear = newNode;
  }
}
```



Enqueue('A')

queuetype.cpp

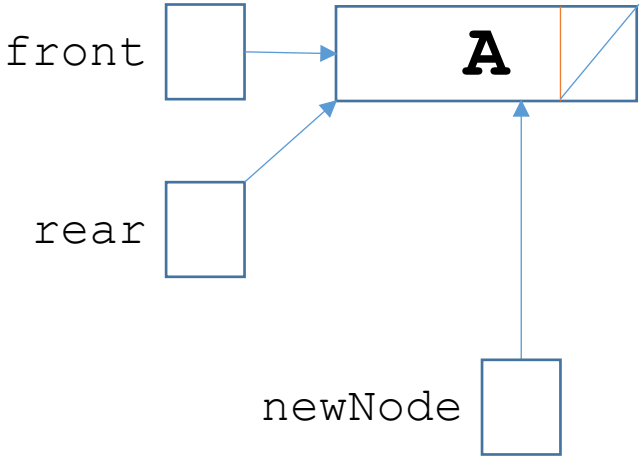
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('A')

queuetype.cpp

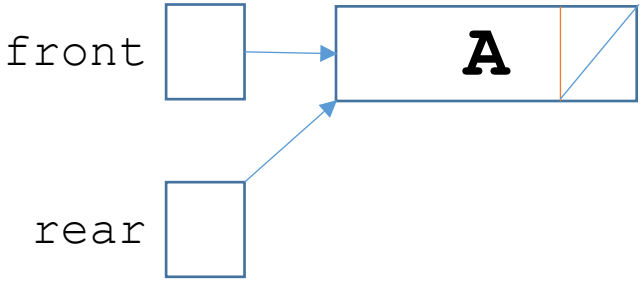
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('A')

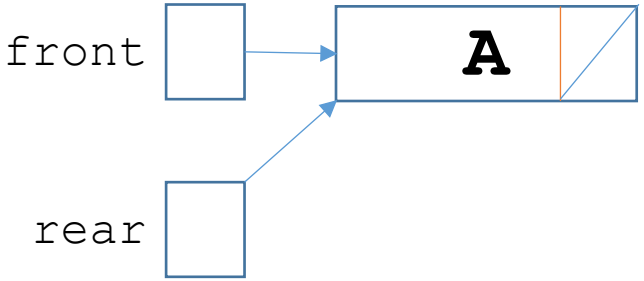
queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



queuetype.cpp

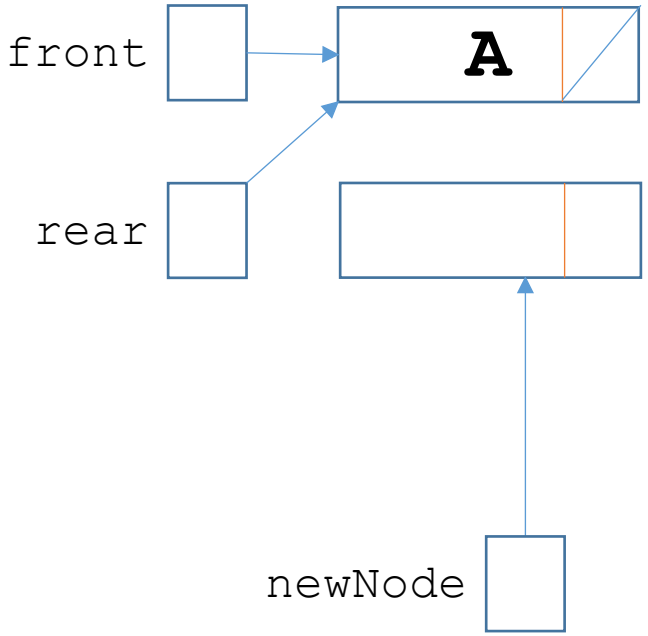
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('B')

queuetype.cpp

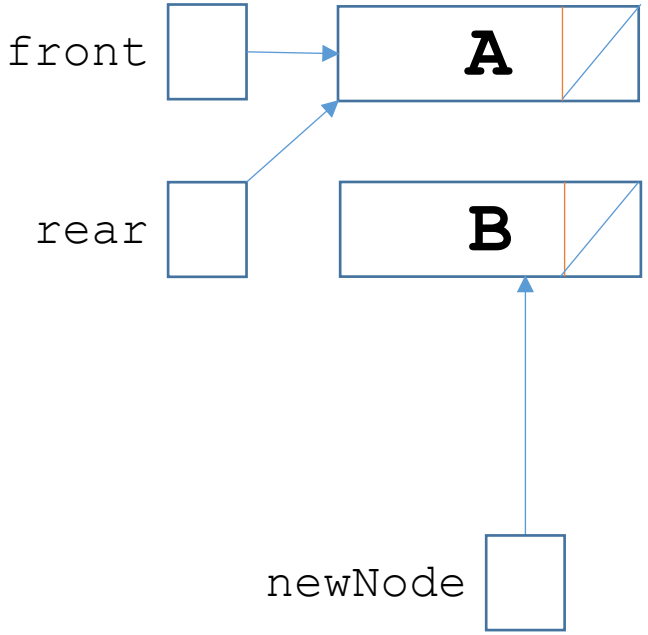
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('B')

queuetype.cpp

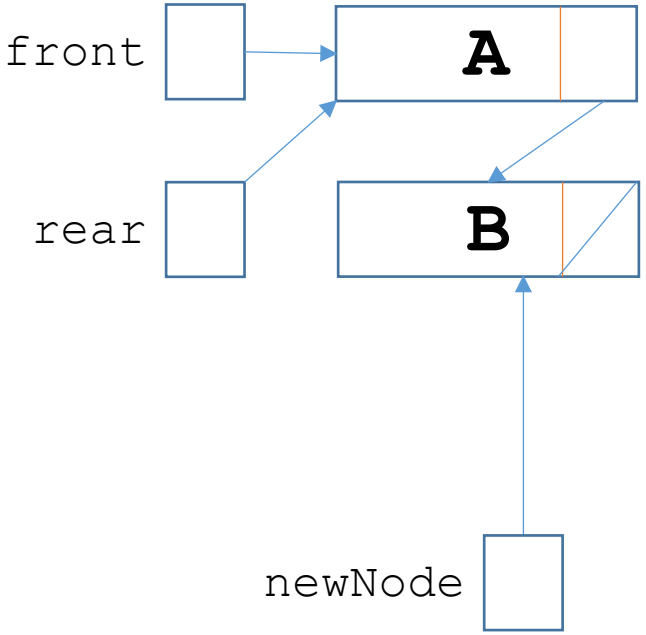
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('B')

queuetype.cpp

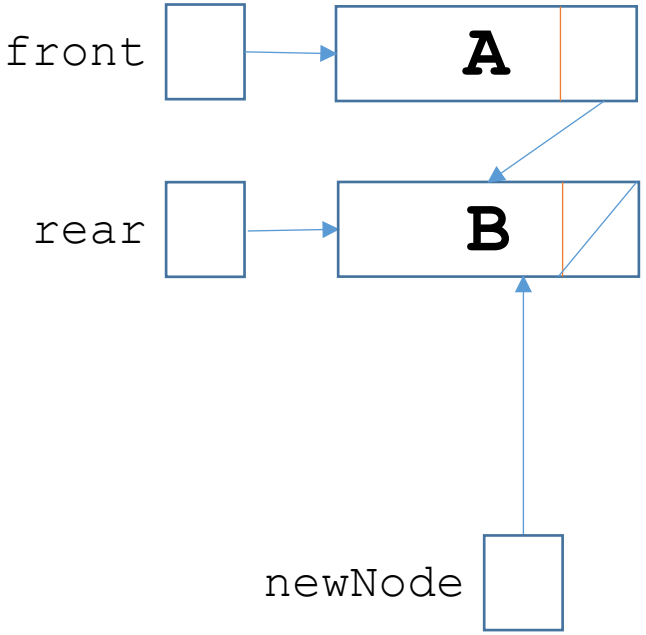
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('B')

queuetype.cpp

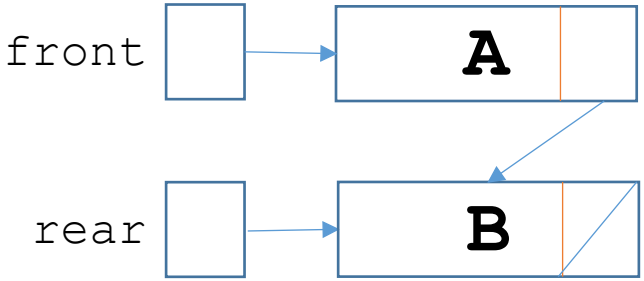
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('B')

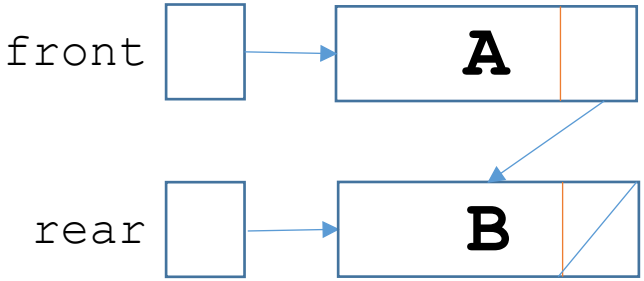
queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



queuetype.cpp

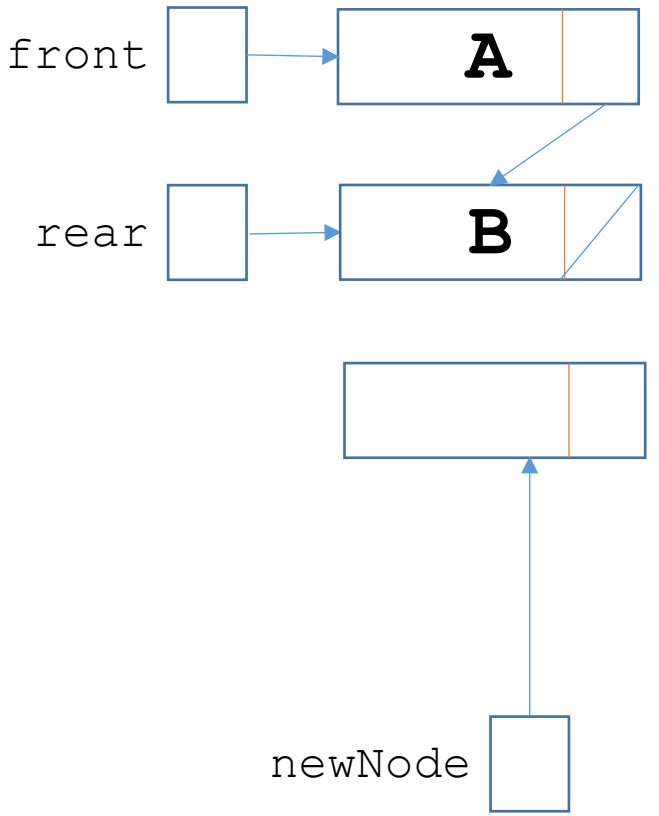
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('C')

queuetype.cpp

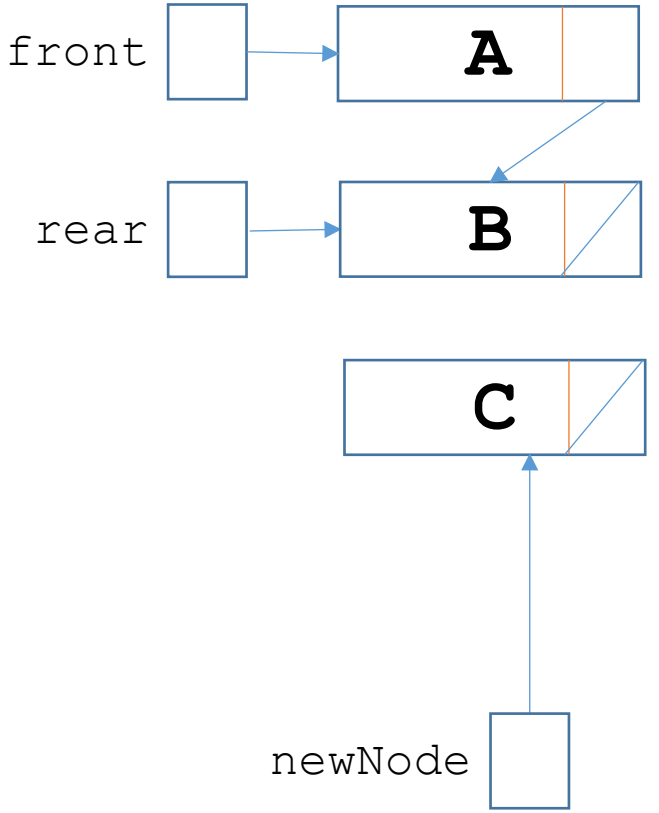
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('C')

queuetype.cpp

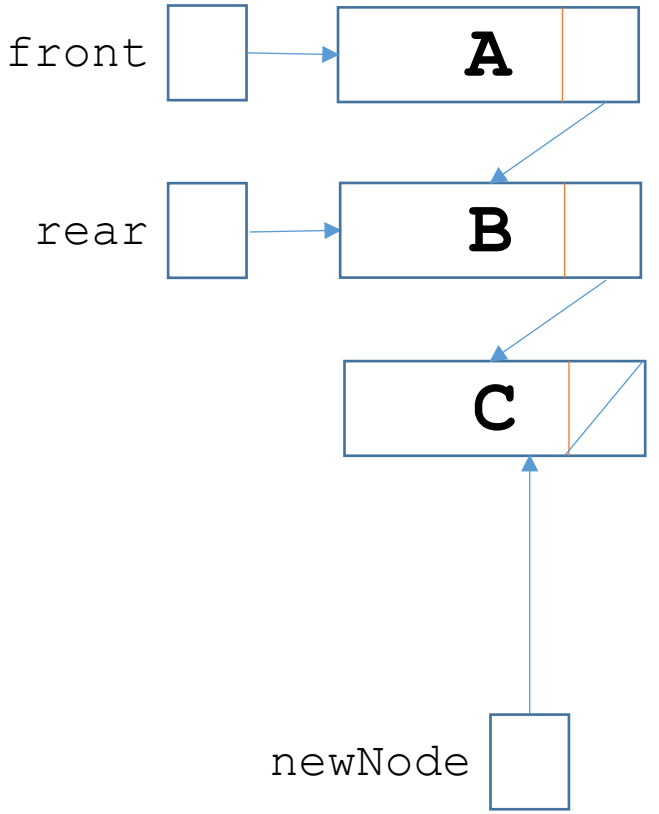
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('C')

queuetype.cpp

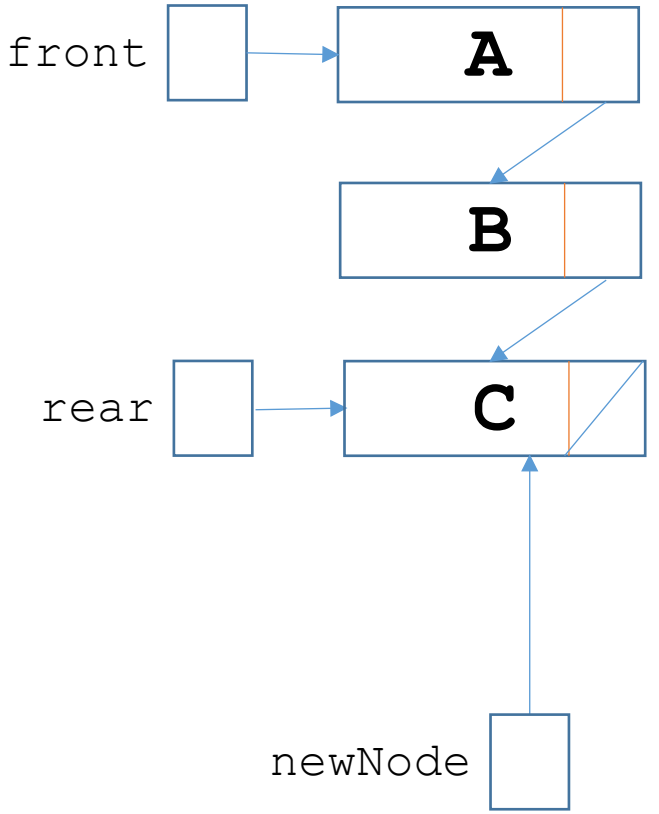
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('C')

queuetype.cpp

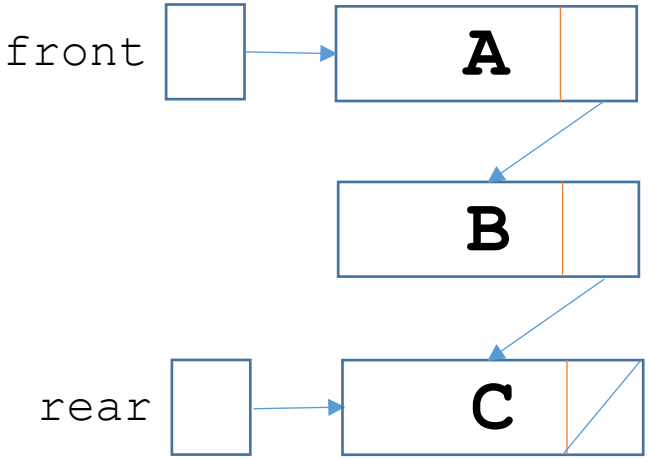
```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



Enqueue('C')

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```



queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```

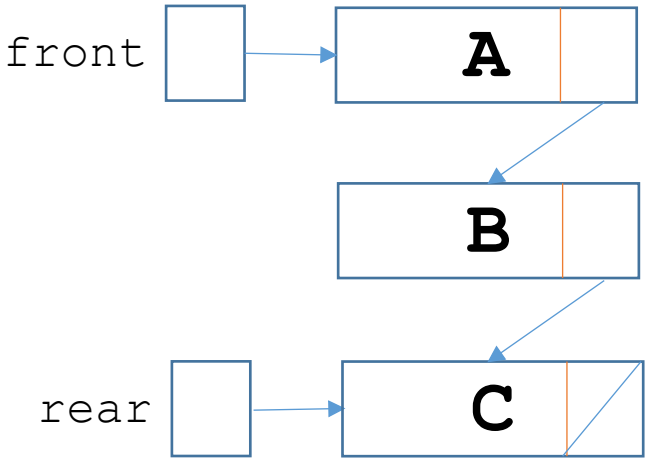
O(1)

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```

queuetype.cpp

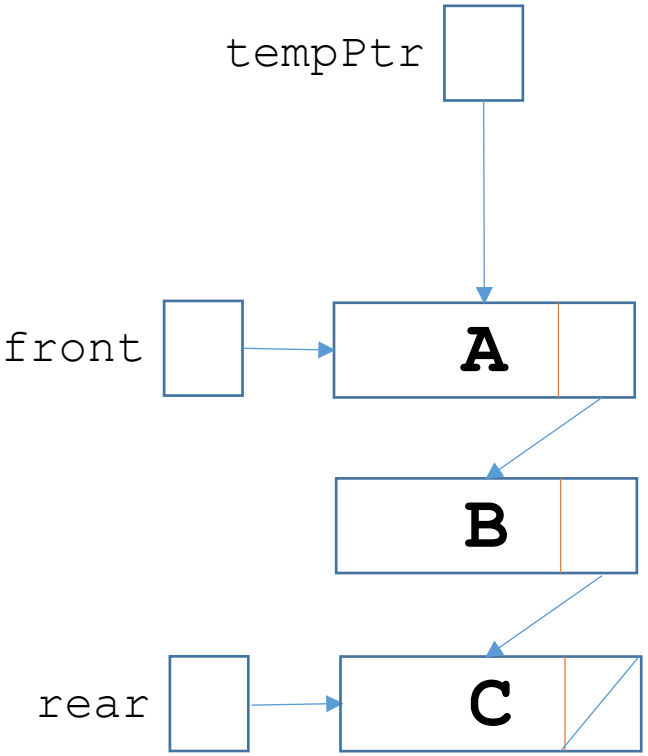
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

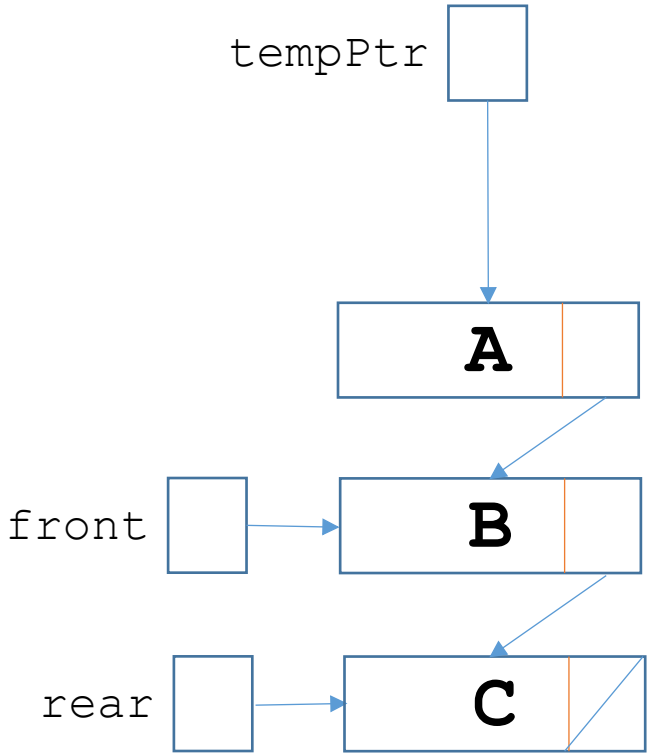
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

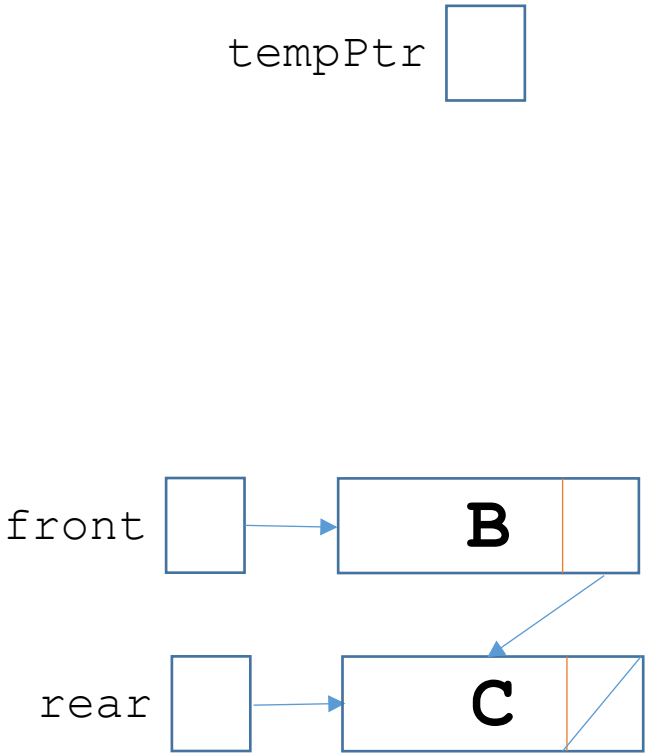
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

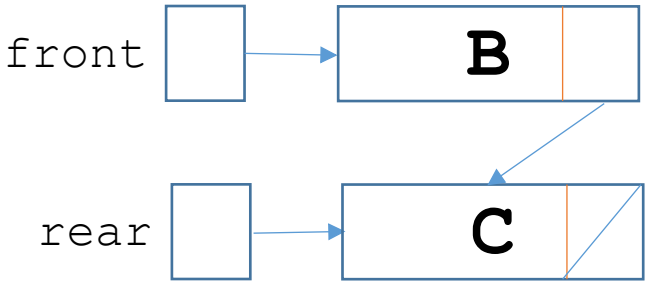
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

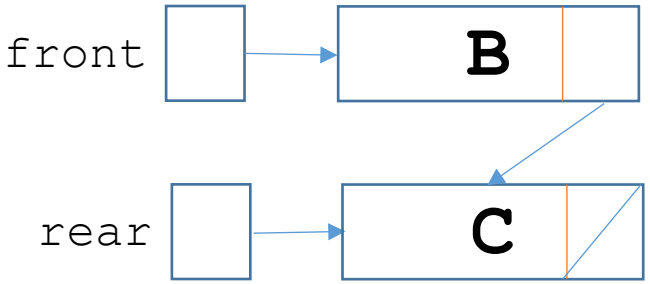
queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



queuetype.cpp

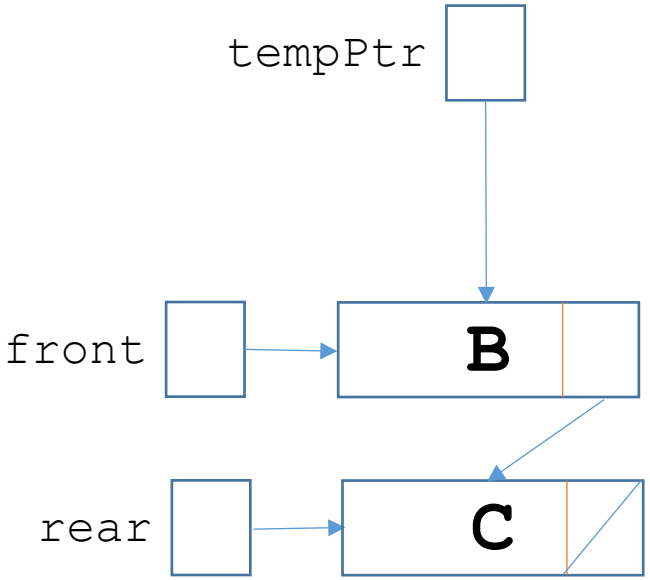
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

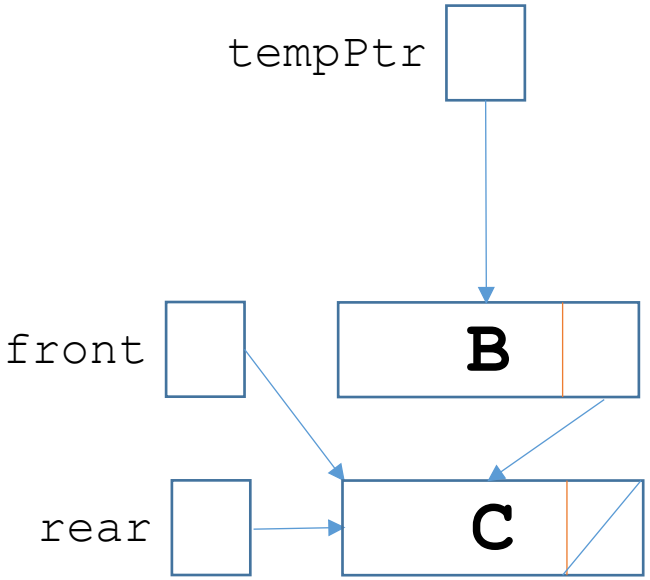
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

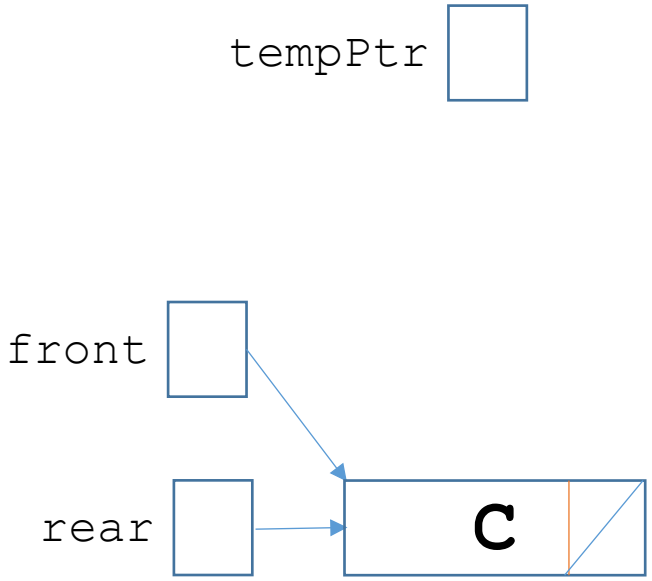
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

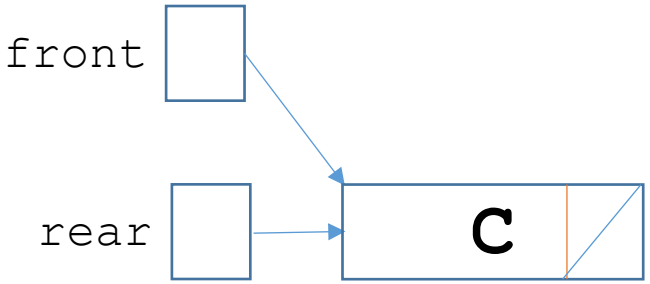
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

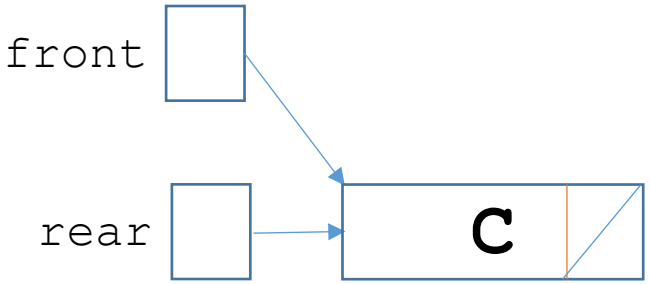
queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



queuetype.cpp

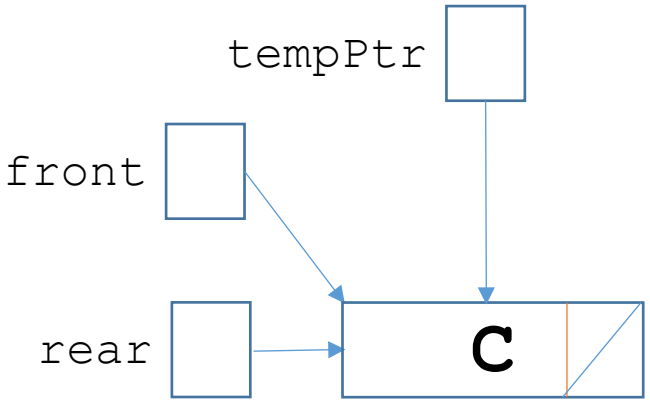
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

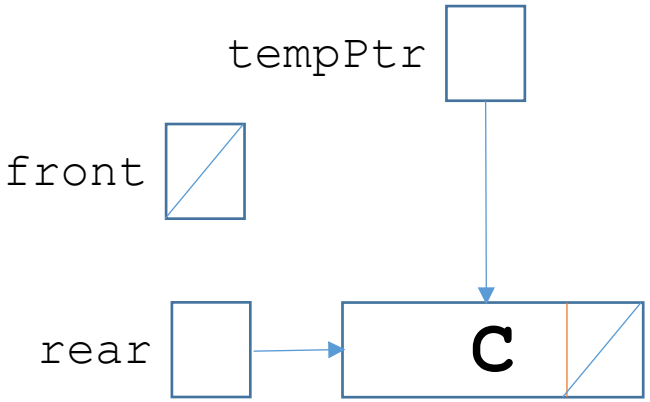
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

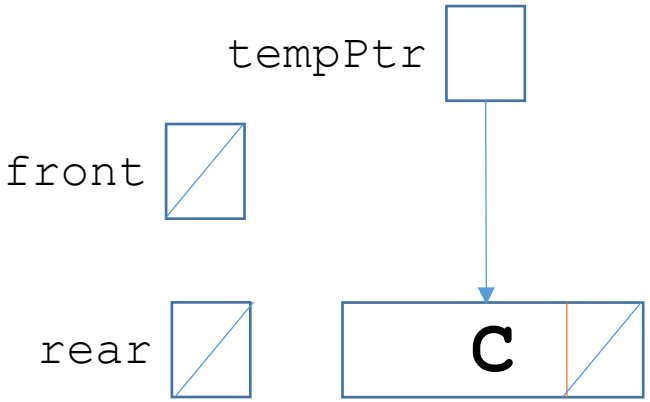
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

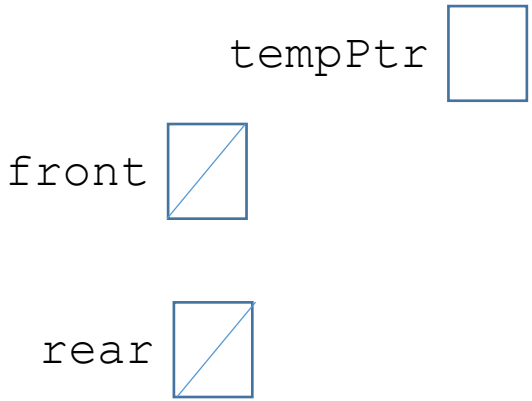
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

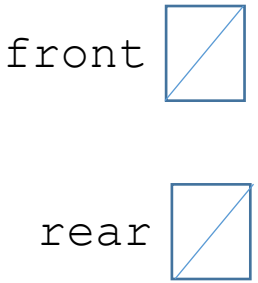
```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



Dequeue()

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```



queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```

O(1)

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::MakeEmpty()
{
  NodeType* tempPtr;

  while (front != NULL)
  {
    tempPtr = front;
    front = front->next;
    delete tempPtr;
  }
  rear = NULL;
}

template <class ItemType>
QueType<ItemType>::~QueType()
{
  MakeEmpty();
}
```

queuetype.cpp

```
template <class ItemType>
void QueType<ItemType>::MakeEmpty()
{
  NodeType* tempPtr;

  while (front != NULL)
  {
    tempPtr = front;
    front = front->next;
    delete tempPtr;
  }
  rear = NULL;
}
```

O(N)

```
template <class ItemType>
QueType<ItemType>::~QueType()
{
  MakeEmpty();
}
```

O(N)