ALGORITHMS AND DATA STRUCTURES

LINEAR STRUCTURES

LINEAR STRUCTURES

- are abstract data types
- can be used when implementing `stack`, `queue`, and `vector`
- possess elements arranged in a linear fashion
- have an array-like order (linearly indexable)
- **Examples**
  1. static and dynamic arrays
  2. linked lists

ALGORITHMS AND DATA STRUCTURES

INTERMISSION: TEMPLATES

INTERMISSION: TEMPLATES!

- Polymorphism: the ability of using the same code for *different* data types
- C++ implements polymorphism using a construction called ***templates***
- The idea is to extend the concept of *overloading* not just to functions but also to classes
- What we want is to write one piece of code for arbitrary data types

INTERMISSION: TEMPLATES!

- How to write single code for the following?

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
double max(double x, double y) {
    return (x > y) ? x : y;
}
```

INTERMISSION: TEMPLATES!

- How to write single code for the following?

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
double max(double x, double y) {
    return (x > y) ? x : y;
}
```

- Solution: *function templates*

```
template <typename dataType>
dataType max(dataType x, dataType y) {
    return (x > y) ? x : y;
}
```

INTERMISSION: TEMPLATES!

- We now can invoke the function as

```
// integers
max(17, 42);
// double precision
max(3.14159, 2.71828);
// characters
max('A', 'Z');

// even with strings!
max("cat", dog");
```

- All use the same template pattern in order to process such instructions

INTERMISSION: TEMPLATES!

- The template facility doesn't actually save any space
- It generates an entirely new copy of the function that works for that type encountered
- It's not defining a single function that works with many types
- It's instead a pattern from which the compiler can generate specially tailored versions

INTERMISSION: TEMPLATES!

- ***Consequences***:
  - The compiler must have access to template when it sees a call to a template function
  - Prototyping is just not enough!
  - One ***cannot*** separate the interface and implementation then
  - So, the implementation must be available when reading the interface
    - hide details of implementation in separate `.h` file

ALGORITHMS AND DATA STRUCTURES

LINKED LISTS

LINKED LISTS

- Suppose you want to write the alphabet in an array

```
A C D E F G H I J K L M N ... Z
```

- To amend the mistake you have to make an insertion $O(N)$

LINKED LISTS

- Suppose you want to write the alphabet in an array

```
A C D E F G H I J K L M N ... Z
```

- To amend the mistake you have to make an insertion $O(N)$
- Or you could devise a new strategy

```
 B
A C D E F G H I J K L M N ... Z
 ^
```

- where the insertion takes $O(1)$

LINKED LISTS

- Instead of using an array use the following notation

```
        ->E-
       /    \
A->C->D      ->F->G->H->I->J->...Z
```

- where the insertion would be represented as

```
  ->B-         ->E-
 /    \       /    \
A      ->C->D      ->F->G->H->I->J->...Z
```

LINKED LISTS

- How to implement such structure?

LINKED LISTS

- How to implement such structure?
  - Pointers because they can point to other objects
  - Pointers indicate an ordering relationship, they have an arithmetic
  - Pointers are links between such objects

LINKED LISTS

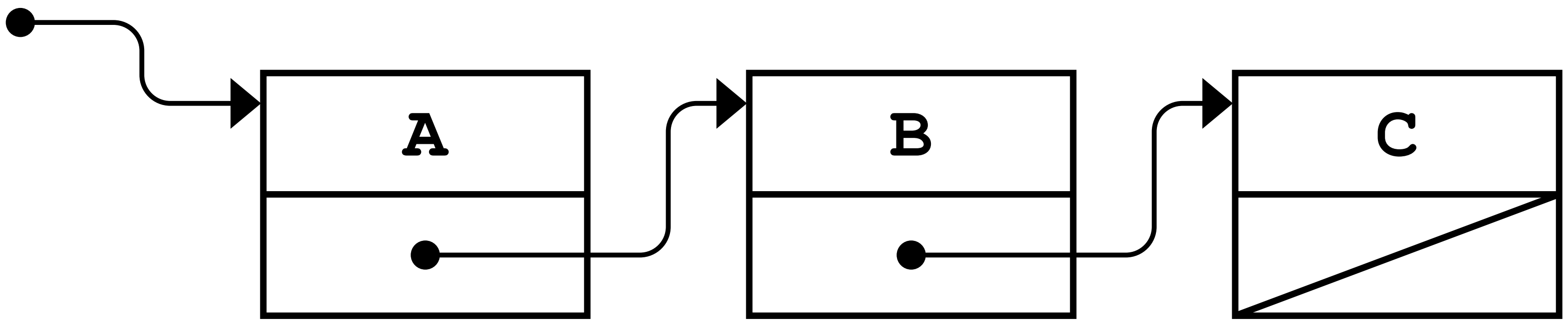
- How to implement such structure?
  - Pointers because they can point to other objects
  - Pointers indicate an ordering relationship, they have an arithmetic
  - Pointers are links between such objects
  - They form a linearly ordered data where each element points to its successor: this is a ***linked list*** (LL)

LINKED LISTS

- What are the basic elements in a LL?
  - Divide the list into a basic element: cells, nodes, or units
  - Have to deal with boundaries to the LL
- Let's say we want to construct a LL of `char`'s

LINKED LISTS

- What are the basic elements in a LL?
  - Divide the list into a basic element: cells, nodes, or units
  - Have to deal with boundaries to the LL
- Let's say we want to construct a LL of `char`'s
- Let's change the previous diagrams to the more useful



LINKED LISTS

- A LL can be, at the basic level, a simple structure

```
struct Node {
    char ch;
    Node *link;
};
```

- Using templates we have

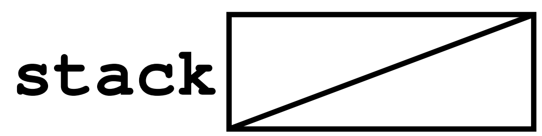
```
template <typename dataType>
struct Cell {
    dataType content;
    Cell *link;
};
```

ALGORITHMS AND DATA STRUCTURES

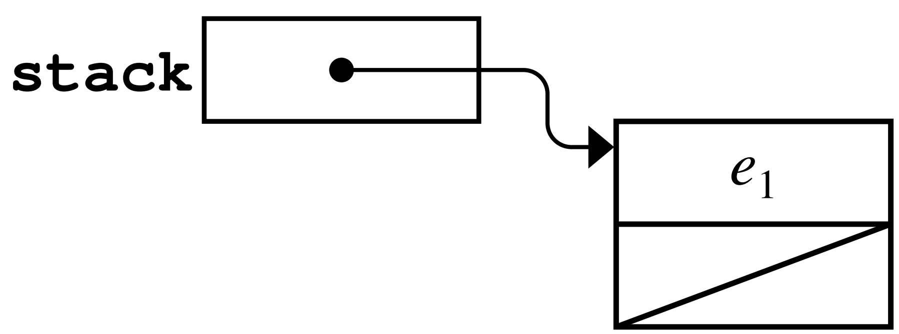
IMPLEMENTING STACK WITH LINKED LISTS

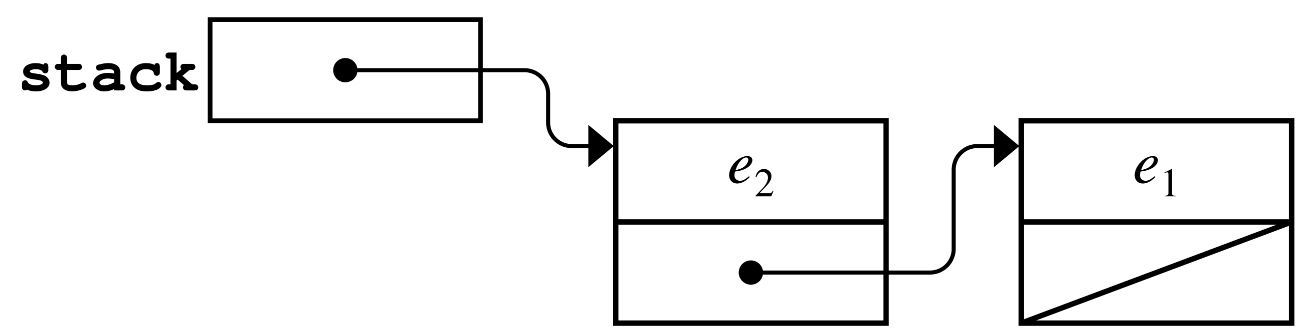
STACKS AS LINKED LISTS

- We use the following drawing convention
  - Empty stack

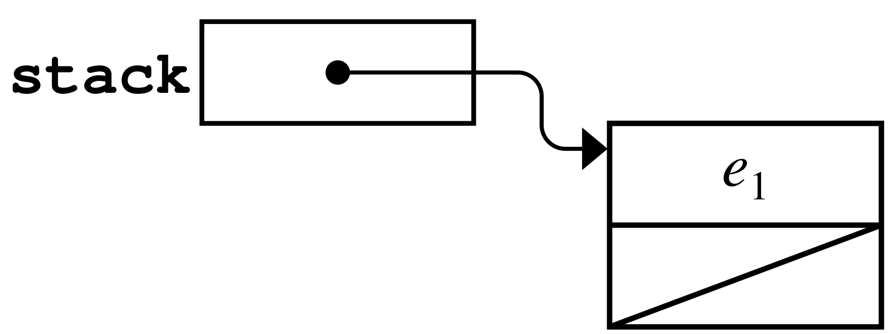


  - Push





  - Pop



```
template <typename dataType>
class Stack {
private:
    /* Type for linked list cell */
    struct Cell {
        dataType data;
        Cell *link;
    };

    Cell *stack; /* Beginning of the list of elements */
    int count;   /* Number of elements in the stack */

    void deepCopy(const Stack<dataType> & src);

   // methods as before

public:
    // methods as before
};
```

```
#include <iostream>
#include "stack.hpp"
using namespace std;

int main() {
    // declaring an instance of the class
    Stack<int> myStack;

    // pusing an integer to stack
    myStack.push(42);

    // removing and printing it
    cout << myStack.pop() << endl;

    return 0;
}
```

ALGORITHMS AND DATA STRUCTURES

IMPLEMENTING QUEUE

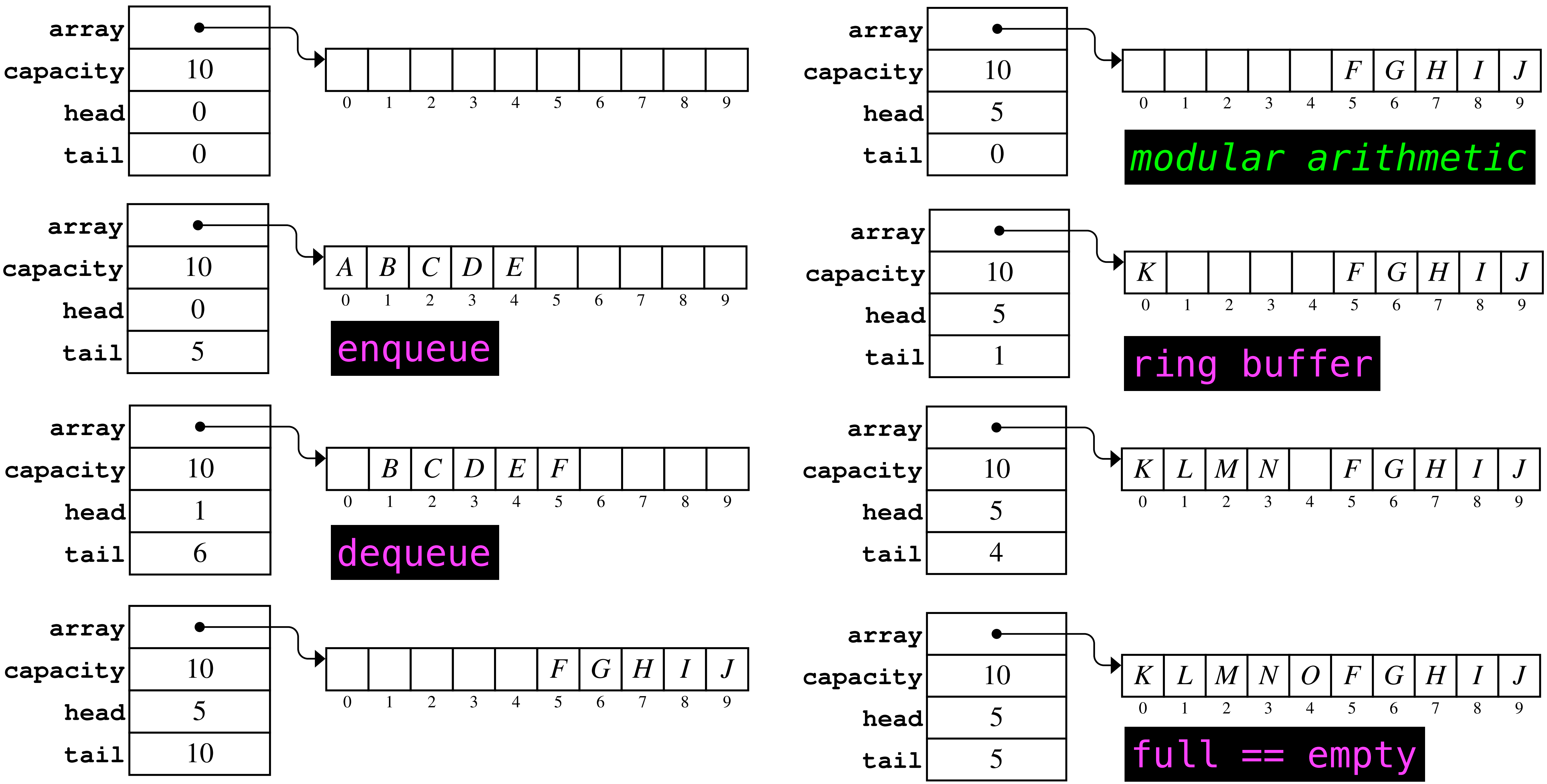
IMPLEMENTING QUEUES

- Structure of implementation very similar to that of `stack`
- Main difference:
  - `push` → `enqueue`
  - `pop` → `dequeue`
  - `peek` → `back` & `???` → `front`
- Can be implemented using arrays or linked list
  - have subtleties that do not appear with stack

ARRAY-BASED REPRESENTATION OF QUEUE

- Needs to keep track of both beginning and end of queue
  - `head`: holds index of next element to leave
  - `tail`: holds index of next free slot
  - `array`: pointer to first element of values
  - `capacity`: contains real size of array
  - `size`: number of elements in container

ARRAY-BASED REPRESENTATION OF QUEUE



```
template <typename dataType>
class Queue {
public:
    Queue();
    ~Queue(){ delete[] array; }
    // size, empty, clear as before
   void enqueue(dataType val);
    dataType dequeue();
    dataType front();
    dataType back();

private:
    dataType *array;
    int capacity, head, tail;
    static const int INITIAL_CAPACITY = 10;

    void deepCopy(const Queue<dataType> & src);
    void expandCapacity();
};
```

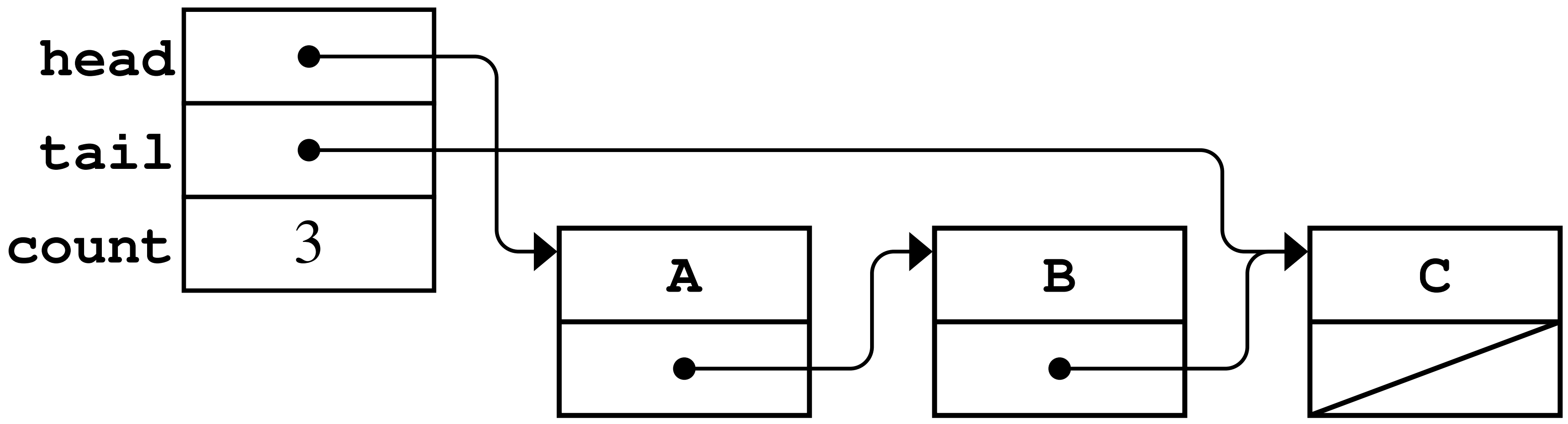
```
template <typename dataType>
Queue<dataType>::Queue() {
   array = new dataType[capacity = INITIAL_CAPACITY];
    head = tail = 0;
}

template <typename dataType>
int Queue<dataType>::size() {
    return (tail + capacity - head) % capacity;
}

template <typename dataType>
bool Queue<dataType>::empty() {
    return head == tail;
}

template <typename dataType>
void Queue<dataType>::clear() {
   head = tail = 0;
}
```

```
template <typename dataType>
void Queue<dataType>::enqueue(dataType elem) {
    if (size() == capacity - 1)
        expandCapacity();

    array[tail] = elem;
    tail = (tail + 1) % capacity;
}

template <typename dataType>
dataType Queue<dataType>::dequeue() {
    if (empty()) error("dequeue: cannot dequeue
                        an empty queue”);

    dataType result = array[head];
    head = (head + 1) % capacity;

    return result;
}
```

```
template <typename dataType>
dataType Queue<dataType>::front() {
    if (empty()) error("front: cannot peek at
                        an empty queue”);

   return array[head];
}

template <typename dataType>
dataType Queue<dataType>::back() {
    if (empty()) error("back: cannot peek at
                        an empty queue”);

   return array[tail];
}
```

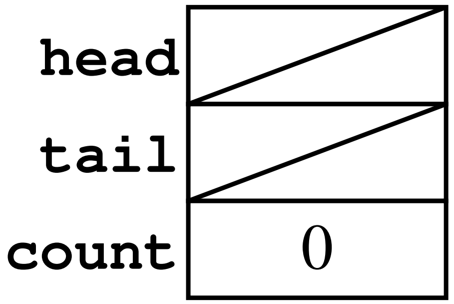
QUEUES IMPLEMENTED WITH LINKED LISTS

- Elements are stored beginning at the head of the queue
- And end at the tail of the queue
  - We do this keeping two pointers (head and `tail`)
  - **Example** with three elements:



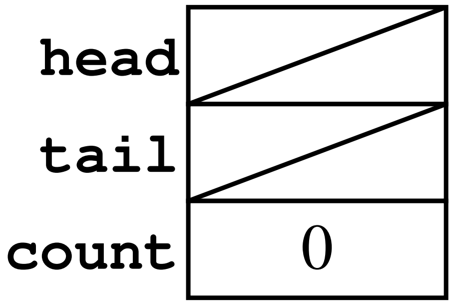
QUEUES IMPLEMENTED WITH LINKED LISTS

`empty queue`

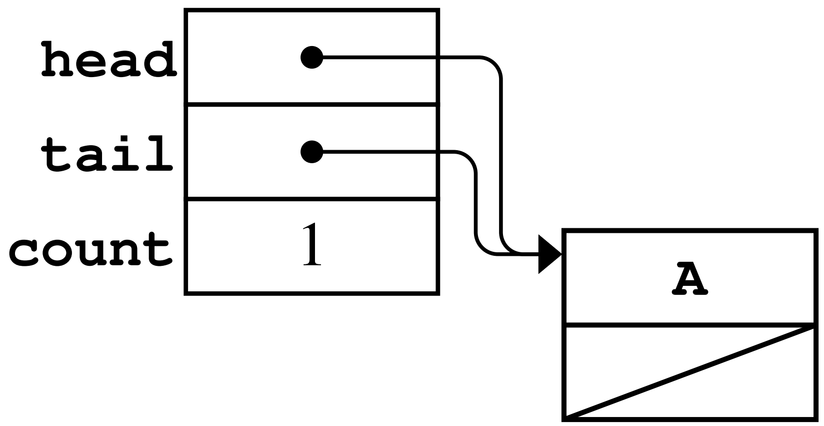


QUEUES IMPLEMENTED WITH LINKED LISTS

`empty queue`

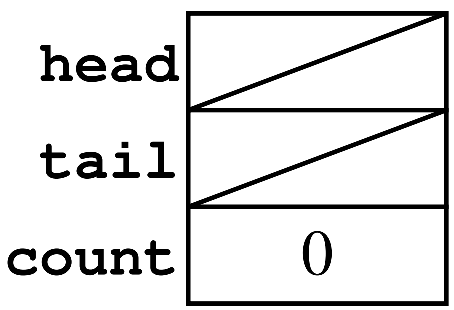


`enqueue empty queue`

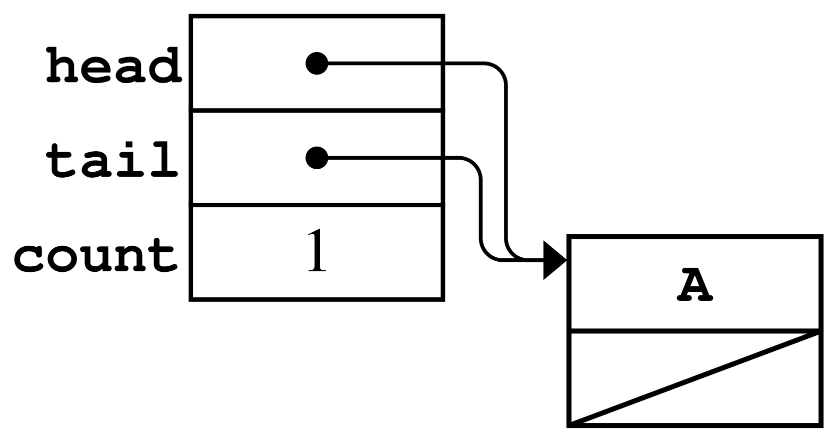


QUEUES IMPLEMENTED WITH LINKED LISTS

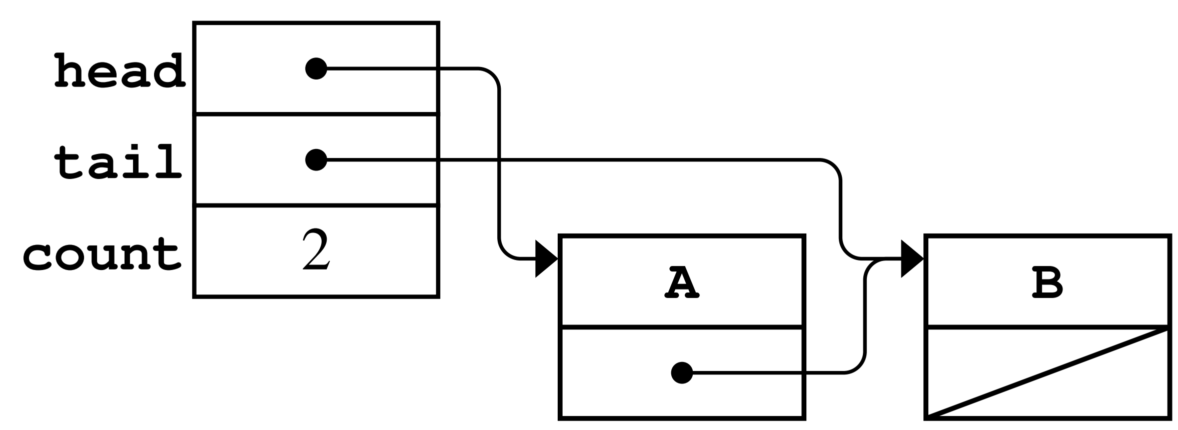
empty queue



enqueue empty queue



enqueue non-empty queue



```
private:
    struct Cell {
        dataType data;
        Cell *link;
    };

    Cell *head;
    Cell *tail;
    int count;

    void deepCopy(const Queue<dataType> & src);
    Queue(const Queue & val) { }
    const Queue & operator=(const Queue & rhs)
    { return *this; }
```

```
template <typename dataType>
Queue<dataType>::Queue() {
   head = tail = nullptr; // NULL for non C++11
    count = 0;
}

template <typename dataType>
Queue<dataType>::~Queue() {
    clear();
}

template <typename dataType>
int Queue<dataType>::size() {
    return count;
}

template <typename dataType>
bool Queue<dataType>::empty() {
    return count == 0;
}
```

```
template <typename dataType>
void Queue<dataType>::clear() {
    while (count > 0) {
        dequeue();
    }
}

template <typename dataType>
dataType Queue<dataType>::front() {
    if (empty()) error("front: peeking at an
                       empty queue");
    return head->data;
}

template <typename dataType>
dataType Queue<dataType>::back() {
    if (empty()) error("back: peeking at an
                       empty queue");
    return tail->data;
}
```

```
template <typename dataType>
void Queue<dataType>::enqueue(dataType elem) {
    Cell *cell = new Cell;
    cell->data = elem;
    // use NULL for non C++11
    cell->link = nullptr;

    // use NULL for non C++11
    if (head == nullptr) {
        head = cell;
    } else {
        tail->link = cell;
    }

    tail = cell;
    count++;
}
```

```
template <typename dataType>
dataType Queue<dataType>::dequeue() {
    if (empty()) error("dequeue: dequeuing
                        an empty queue");
    Cell *cell = head;
    dataType result = cell->data;
    head = cell->link;

    // use NULL for non C++11
    if (head == nullptr)
        tail = nullptr;

    count--;
    delete cell;

    return result;
}
```

```
template <typename T>
void Queue<T>::deepCopy(const Queue<T> & src) {
    head == nullptr
    tail == nullptr
    count = 0;

    Cell *ip;
    for (ip = src.head; ip != nullptr; ip = ip->link) {
        enqueue(ip->data);
    }
}
```

EXTRA READING

- From the book Programming Abstractions in C++:
  - 14.2 *Implementing stacks* (array & list implementation)
  - 14.3 *Implementing queues*
  - 14.4 *Implementing vectors*

Roberts, Eric. (2013). *Programming Abstractions in C++*. Pearson.