Data Structure and Algorithms [CO2003]

Chapter 5 - Stack and Queue

Lecturer: Duc Dung Nguyen, PhD.
Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

Contents

Outcomes

- **L.O.2.1** - Depict the following concepts: (a) array list and linked list, including single link and double links, and multiple links; (b) stack; and (c) queue and circular queue.
- **L.O.2.2** - Describe storage structures by using pseudocode for: (a) array list and linked list, including single link and double links, and multiple links; (b) stack; and (c) queue and circular queue.
- **L.O.2.3** - List necessary methods supplied for list, stack, and queue, and describe them using pseudocode.
- **L.O.2.4** - Implement list, stack, and queue using C/C++.

Outcomes

- **L.O.2.5** - Use list, stack, and queue for problems in real-life, and choose an appropriate implementation type (array vs. link).
- **L.O.2.6** - Analyze the complexity and develop experiment (program) to evaluate the efficiency of methods supplied for list, stack, and queue.
- **L.O.8.4** - Develop recursive implementations for methods supplied for the following structures: list, tree, heap, searching, and graphs.
- **L.O.1.2** - Analyze algorithms and use Big-O notation to characterize the computational complexity of algorithms composed by using the following control structures: sequence, branching, and iteration (not recursion).

Basic operations of Stacks

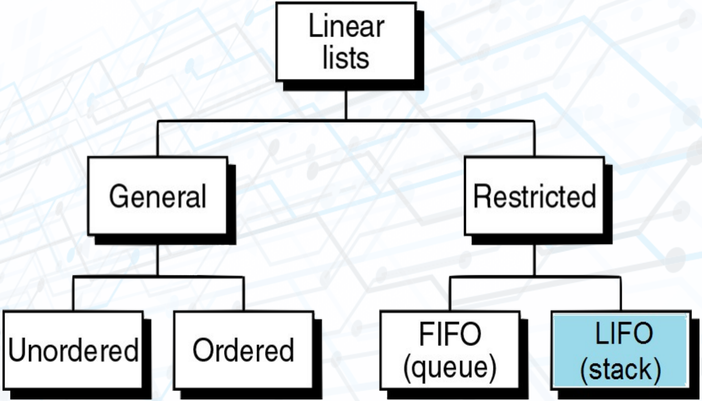
General list:

- No restrictions on which operation can be used on the list.
- No restrictions on where data can be inserted/deleted.

Restricted list:

- Only some operations can be used on the list.
- Data can be inserted/deleted only at the ends of the list.

Linear list concepts



Definition
A stack of elements of type T is a finite sequence of elements of T, in which all insertions and deletions are restricted to one end, called the top.

Stack is a Last In - First Out (LIFO) data structure.
LIFO: The last item put on the stack is the first item that can be taken off.

Basic operations:

- `Construct` a stack, leaving it empty.
- `Push` an element: put a new element on to the top of the stack.
- `Pop` an element: remove the top element from the top of the stack.
- `Top` an element: retrieve the top element.

Extended operations:

- Determine whether the stack is `empty` or not.
- Determine whether the stack is `full` or not.
- Find the `size` of the stack.
- `Clear` the stack to make it empty.

Basic operations of Stacks: Push

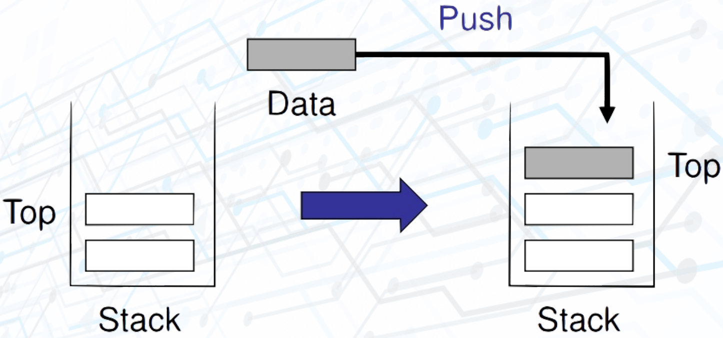


Figure 1: Successful Push operation

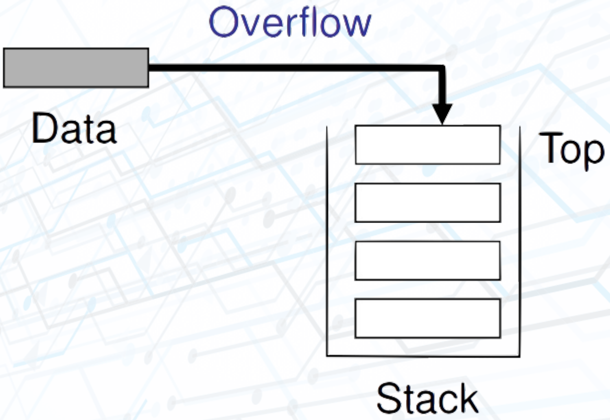


Figure 2: Unsuccessful Push operation. Stack remains unchanged.

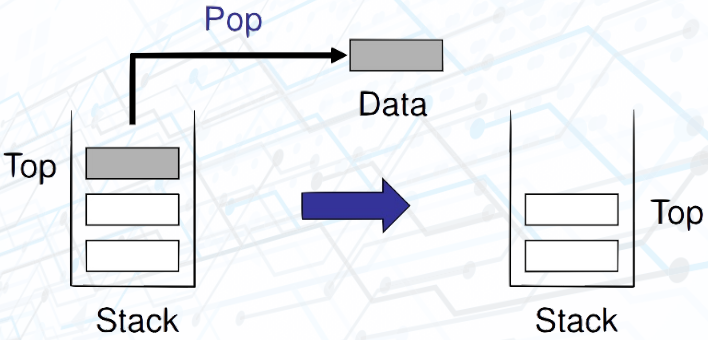


Figure 3: Successful Pop operation

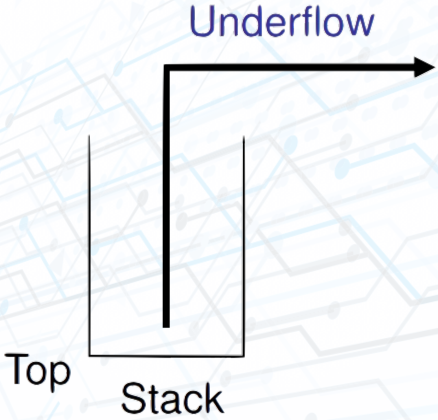


Figure 4: Unsuccessful Pop operation. Stack remains unchanged.

Basic operations of Stacks: Top

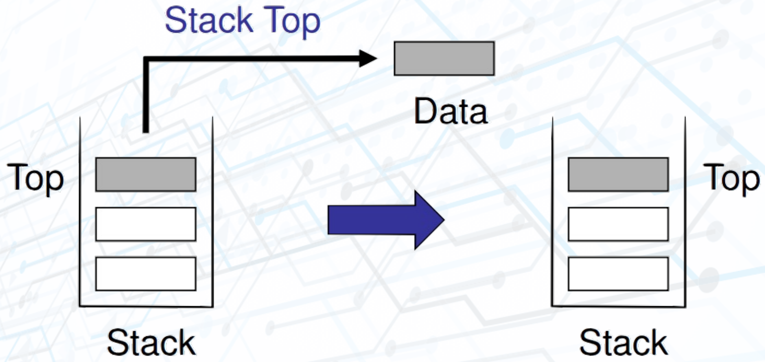


Figure 5: Successful Top operation. Stack remains unchanged.

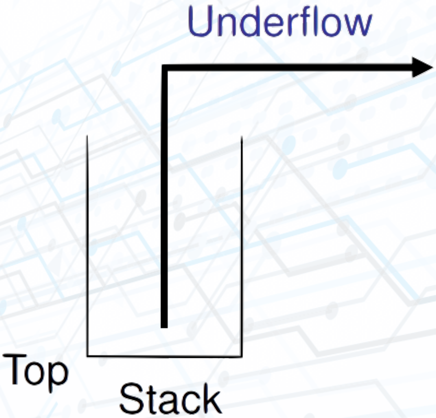
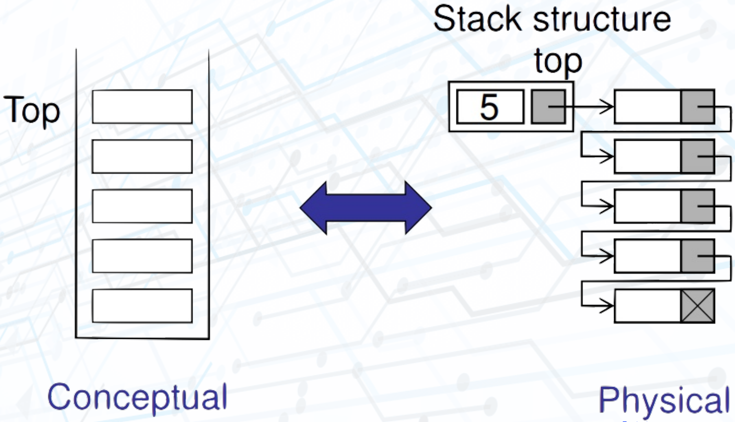


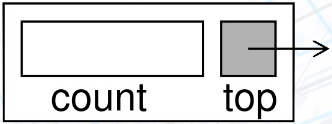
Figure 6: Unsuccessful Top operation. Stack remains unchanged.

Implementation of Stacks



throw std::runtime_error("Error");

Stack structure




```
stack
   count <integer>
   top <node pointer>
end stack
```

Stack node structure


```
node
   data <dataType>
   next <node pointer>
end node
```

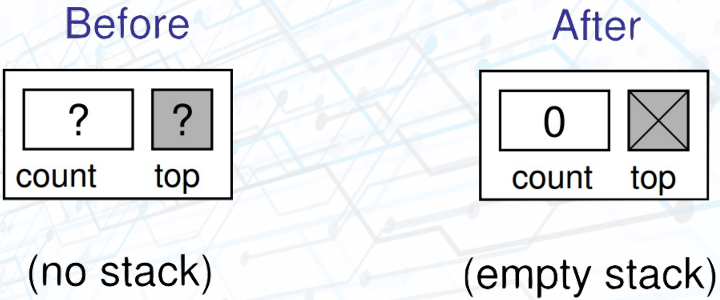
Linked-list implementation in C++

```cpp
template <class ItemType>
struct Node {
  ItemType data;
  Node<ItemType> *next;
};
```

Linked-list implementation in C++

```
template <class List_ItemType>
class Stack {
  public:
    Stack();
    ~Stack();
    void Push(List_ItemType dataIn);
    int  Pop(List_ItemType &dataOut);
    int  GetStackTop(List_ItemType &dataOut);
    void Clear();
    int  IsEmpty();
    int  GetSize();
    Stack<List_ItemType>* Clone();
    void Print2Console();
  private:
    Node<List_ItemType>* top;
    int count;
};
```

Create an empty Linked Stack



Create an empty Linked Stack

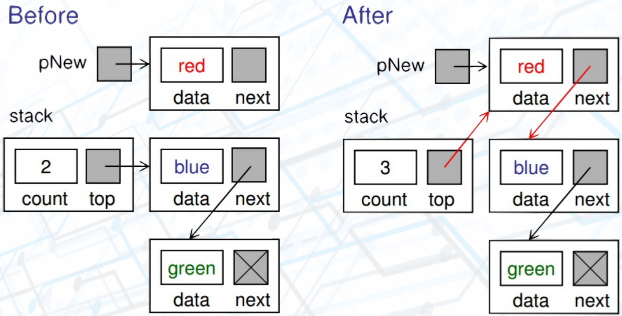
Algorithm createStack(ref stack <metadata>)
Initializes the metadata of a stack
Pre: `stack` is a metadata structure of a stack
Post: metadata initialized

stack.count = 0
stack.top = null
return
End createStack

Create an empty Linked Stack

```
template <class List_ItemType>
Stack<List_ItemType>::Stack(){
  this->top = NULL;
  this->count = 0;
}

template <class List_ItemType>
Stack<List_ItemType>::~Stack(){
  this->Clear();
}
```

Push data into a Linked Stack



1. Allocate memory for the new node and set up data.
2. Update pointers:
   - Point the new node to the top node (before adding the new node).
   - Point top to the new node.
3. Update count

Push data into a Linked Stack

Algorithm pushStack(ref stack <metadata>, val data <dataType>)
Inserts (pushes) one item into the stack
Pre: `stack` is a metadata structure to a valid stack
`data` contains value to be pushed into the stack
Post: data have been pushed in stack
Return `true` if successful; `false` if memory overflow

Push data into a Linked Stack

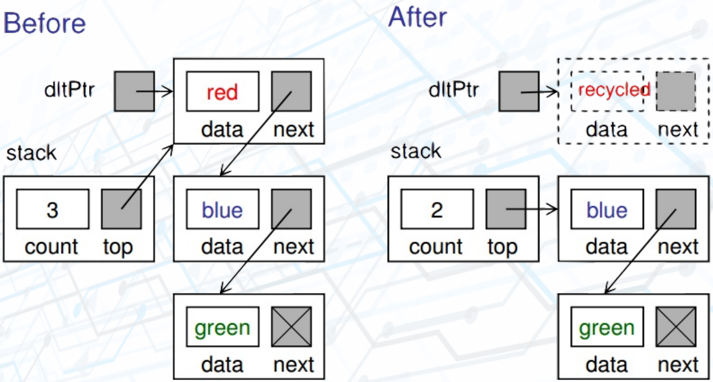
```
if stack full then
    success = false
else
    allocate (pNew)
    pNew -> data = data
    pNew -> next = stack.top
    stack.top = pNew
    stack.count = stack.count + 1
    success = true
end
return success
End pushStack
```

Push data into a Linked Stack

```
template <class List_ItemType>
void Stack<List_ItemType>::Push
                (List_ItemType value){
  Node<List_ItemType>* pNew =
                new Node<List_ItemType>();
  pNew->data = value;
  pNew->next = this->top;
  this->top = pNew;
  this->count++;
}
```

Push data into a Linked Stack

- Push is successful when allocation memory for the new node is successful.
- There is no difference between push data into a stack having elements and push data into an empty stack (`top` having NULL value is assigned to `pNew->next`: that's corresponding to a list having only one element).



1. `dltPtr` holds the element on the top of the stack.
2. `top` points to the next element.
3. Recycle `dltPtr`. Decrease `count` by 1.

Pop Linked Stack

Algorithm popStack(ref stack <metadata>, ref dataOut <dataType>)

Pops the item on the top of the stack and returns it to caller

Pre: `stack` is a metadata structure to a valid stack

`dataOut` is to receive the popped data

Post: data have been returned to caller

Return `true` if successful; `false` if stack is empty

Pop Linked Stack

if *stack empty* **then**
  success = false
else
  dltPtr = stack.top
  dataOut = stack.top -> data
  stack.top = stack.top -> next
  stack.count = stack.count - 1
  recycle(dltPtr)
  success = true
end
return success
End popStack

```
template <class List_ItemType>
int Stack<List_ItemType>::Pop
                (List_ItemType &dataOut){
  if (this->GetSize() == 0)
    return 0;
  Node<List_ItemType>* dltPtr = this->top;
  dataOut = dltPtr->data;
  this->top = dltPtr->next;
  this->count--;
  delete dltPtr;
  return 1;
}
```

Pop Linked Stack

- Pop is successful when the stack is not empty.
- There is no difference between pop an element from a stack having elements and pop the only-one element in the stack (`dltPtr->next` having NULL value is assigned to top: that's corresponding to an empty stack).

Stack Top

Algorithm stackTop(ref stack <metadata>, ref dataOut <dataType>)

Retrieves the data from the top of the stack without changing the stack

Pre: `stack` is a metadata structure to a valid stack

`dataOut` is to receive top stack data

Post: data have been returned to caller

Return `true` if successful; `false` if stack is empty

```
if stack empty then
    success = false
else
    dataOut = stack.top -> data
    success = true
end
return success
End stackTop
```

Stack Top

```
template <class List_ItemType>
int Stack<List_ItemType>::GetStackTop
                        (List_ItemType &dataOut){

  if (this->GetSize() == 0)
    return 0;

  dataOut = this->top->data;

  return 1;
}
```

Destroy Stack

Algorithm destroyStack(ref stack <metadata>)
Releases all nodes back to memory

Pre: `stack` is a metadata structure to a valid stack
Post: stack empty and all nodes recycled

Destroy Stack

```
if stack not empty then
    while stack.top not null do
        temp = stack.top
        stack.top = stack.top -> next
        recycle(temp)
    end
end
stack.count = 0
return
End destroyStack
```

```
template <class List_ItemType>
void Stack<List_ItemType>::Clear() {
  Node<List_ItemType>* temp;
  while (this->top != NULL){
    temp = this->top;
    this->top = this->top->next;
    delete temp;
  }
  this->count = 0;
}
```

isEmpty Linked Stack

Algorithm isEmpty(ref stack <metadata>)

Determines if the stack is empty

Pre: `stack` is a metadata structure to a valid stack

Post: return stack status

Return `true` if the stack is empty, `false` otherwise

if *count* $= 0$ **then**
  Return `true`
else
  Return `false`
end

End isEmpty

isEmpty Linked Stack

```
template <class List_ItemType>
int Stack<List_ItemType>::IsEmpty() {
  return (count == 0);
}

template <class List_ItemType>
int Stack<List_ItemType>::GetSize() {
  return count;
}
```

isFull Linked Stack

```
template <class List_ItemType>
int Stack<List_ItemType>::IsFull() {
  Node<List_ItemType>* pNew =
        new Node<List_ItemType>();

  if (pNew != NULL) {
    delete pNew;
    return 0;
  } else {
    return 1;
  }
}
```

Print Stack

```
template <class List_ItemType>
void Stack<List_ItemType>::Print2Console() {
  Node<List_ItemType>* p;
  p = this->top;
  while (p != NULL){
    cout << p->data << "␣";
    p = p->next;
  }
  cout << endl;
}
```

```
int main(int argc, char* argv[]){
    Stack<int> *myStack = new Stack<int>();
    int val;
    myStack->Push(7);
    myStack->Push(9);
    myStack->Push(10);
    myStack->Push(8);
    myStack->Print2Console();
    myStack->Pop(val);
    myStack->Print2Console();
    delete myStack;
    return 0;
}
```

Array-based stack implementation

Implementation of array-based stack is very simple. It uses `top` variable to point to the topmost stack's element in the array.

1. Initialy `top = -1`;
2. `push` operation increases `top` by one and writes pushed element to `storage[top]`;
3. `pop` operation checks that `top` is not equal to -1 and decreases `top` variable by 1;
4. `getTop` operation checks that `top` is not equal to -1 and returns `storage[top]`;
5. `isEmpty` returns boolean if `top == -1`.

Array-based stack implementation

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

class ArrayStack {
private:
  int top;
  int capacity;
  int *storage;
public:
  ArrayStack(int capacity) {
    storage = new int[capacity];
    this->capacity = capacity;
    top = -1;
  }
  // ...
```

```
~ArrayStack() {
  delete[] storage;
}

void push(int value) {
  if (top == capacity - 1)
    throw string("Stack is overflow");
  top++;
  storage[top] = value;
}
void pop(int &dataOut) {
  if (top == -1)
    throw string("Stack is empty");
 dataOut = storage[top];
 top--;
}

// ...
```

Array-based stack implementation

```
int getTop() {
  if (top == -1)
    throw string("Stack␣is␣empty");
  return storage[top];
}

bool isEmpty() {
  return (top == -1);
}

bool isFull() {
  return (top == capacity -1);
}
```

Array-based stack implementation

```
int getSize() {
  return top + 1;
}

void print2Console() {
  if (top > -1) {
    for (int i = top; i >= 0; i--) {
      cout << storage[i] << " ";
    }
    cout << endl;
  }
}

};
```

Using array-based stack

```
int main(int argc, char* argv[]){
  ArrayStack *myStack = new ArrayStack(10);
  int val;
  myStack->push(7);
  myStack->push(9);
  myStack->push(10);
  myStack->push(8);
  myStack->print2Console();
  myStack->pop(val);
  myStack->print2Console();
  delete myStack;
  return 0;
}
```

Applications of Stack

Applications of Stack

- Reversing data items
  - Reverse a list
  - Convert Decimal to Binary
- Parsing
  - Brackets Parse
- Postponement of processing data items
  - Infix to Postfix Transformation
  - Evaluate a Postfix Expression
- Backtracking
  - Goal Seeking Problem
  - Knight's Tour
  - Exiting a Maze
  - Eight Queens Problem

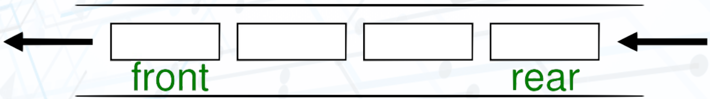
Basic operations of Queues

Definition
A queue of elements of type T is a finite sequence of elements of T, in which data can only be inserted at one end called the rear, and deleted from the other end called the front.

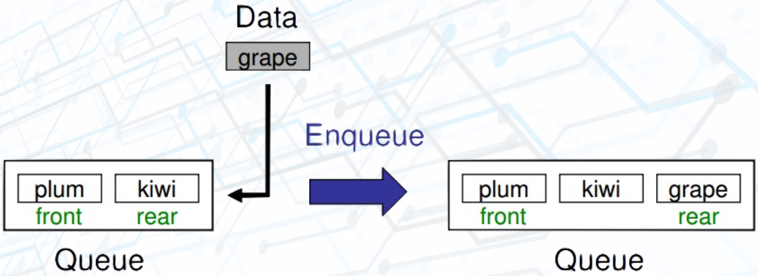
Queue is a First In - First Out (FIFO) data structure.
FIFO: The first item stored in the queue is the first item that can be taken out.

Basic operations:

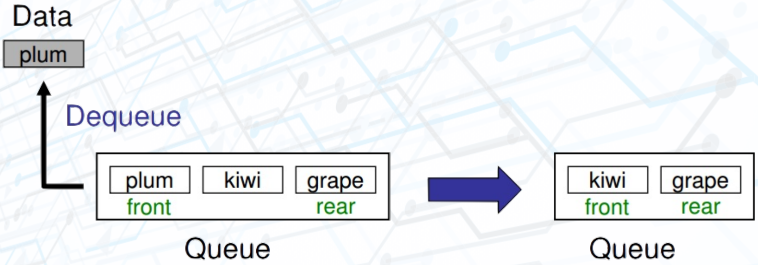
- `Construct` a queue, leaving it empty.
- `Enqueue`: put a new element in to the rear of the queue.
- `Dequeue`: remove the first element from the front of the queue.
- `Queue Front`: retrieve the front element.
- `Queue Rear`: retrieve the rear element.



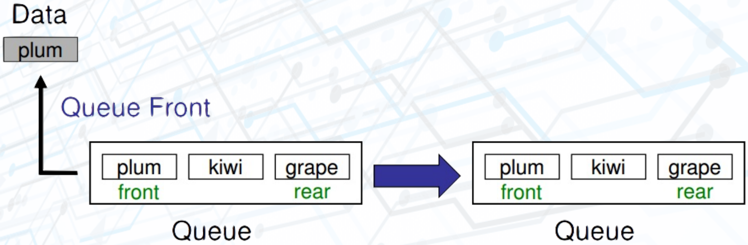
Basic operations of Queues: Enqueue



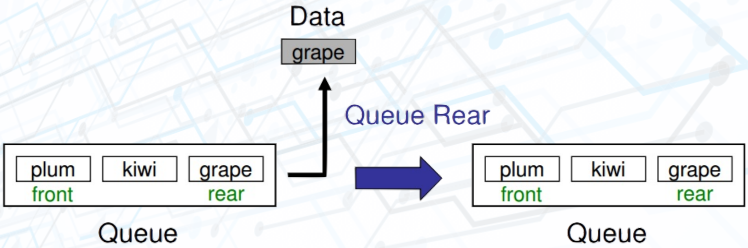
Basic operations of Queues: Dequeue



Basic operations of Queues: Queue Front

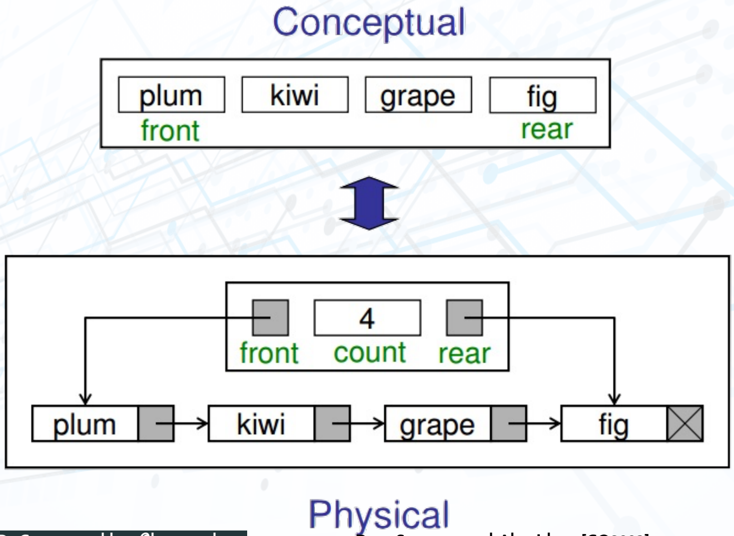


Basic operations of Queues: Queue Rear



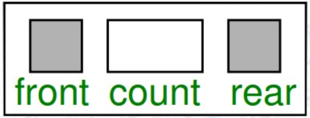
Implementation of Queue

Linked-list implementation



Linked-list implementation

Queue structure




```
queue
   count <integer>
   front <node pointer>
   rear <node pointer>
endqueue
```

Queue node structure


```
node
   data <dataType>
   next <node pointer>
end node
```

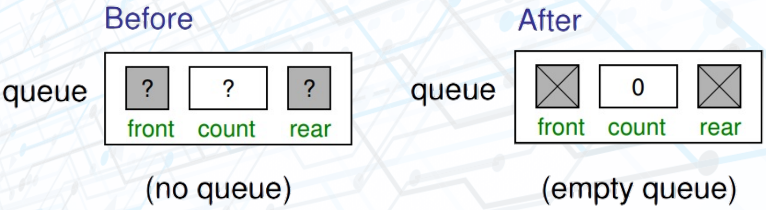
Linked-list implementation in C++

```
template <class ItemType>
struct Node {
  ItemType data;
  Node<ItemType> *next;
};


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

Linked-list implementation in C++

```
    void Enqueue(List_ItemType dataIn);
    int Dequeue(List_ItemType &dataOut);
    int GetQueueFront(List_ItemType &dataOut);
    int GetQueueRear(List_ItemType &dataOut);
    void Clear();
    int IsEmpty();
    int GetSize();
    void Print2Console();

  private:
    Node<List_ItemType> *front, *rear;
    int count;
};
```

Create Queue



Algorithm createQueue(ref queue <metadata>)

Initializes the metadata of a queue

Pre: queue is a metadata structure of a queue

Post: metadata initialized

queue.count= 0

queue.front = null

queue.rear = null

return

End createQueue

Create Queue

```
template <class List_ItemType>
Queue<List_ItemType>::Queue(){
    this->count = 0;
    this->front = NULL;
    this->rear = NULL;
}

template <class List_ItemType>
Queue<List_ItemType>::~Queue(){
    this->Clear();
}
```

Enqueue: Insert into an empty queue

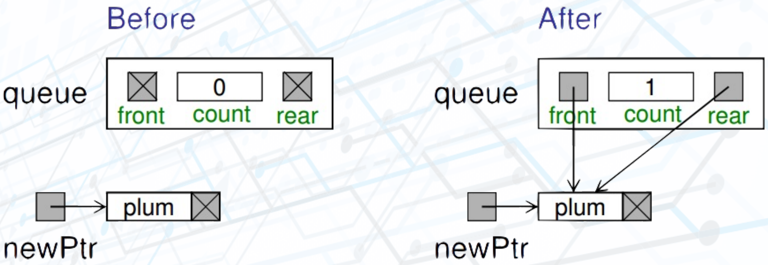


Figure 7: Insert into an empty queue

Enqueue: Insert into a queue with data

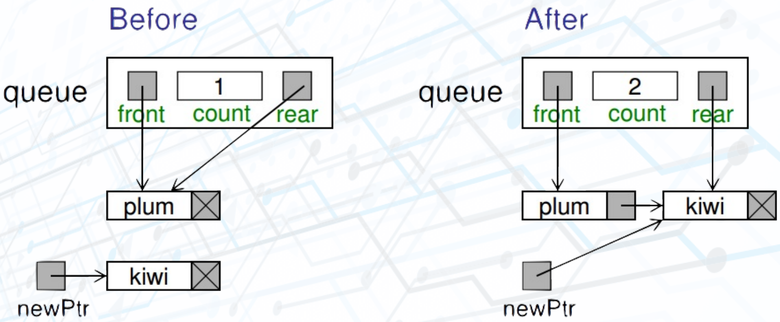


Figure 8: Insert into a queue with data

Enqueue

Algorithm enqueue(ref queue <metadata>, val data <dataType>)
Inserts one item at the rear of the queue

Pre: `queue` is a metadata structure of a valid queue
`data` contains data to be inserted into queue

Post: `data` have been inserted in queue
Return `true` if successful, `false` if memory overflow

Enqueue

```
if queue full then
 |  return false
end
allocate (newPtr)
newPtr -> data = data
newPtr -> next = null
if queue.count = 0 then
 |  queue.front = newPtr // Insert into an empty queue
else
 |  queue.rear -> next = newPtr // Insert into a queue with data
end
queue.rear = newPtr
queue.count = queue.count + 1
return true
```

Enqueue

```
template <class List_ItemType>
void Queue<List_ItemType>::Enqueue
        (List_ItemType value){
  Node<List_ItemType>* newPtr = new Node<List_ItemType>();
  newPtr->data = value;
  newPtr->next = NULL;
  if (this->count == 0)
    this->front = newPtr;
  else
    this->rear->next = newPtr;
  this->rear = newPtr;
  this->count++;
}
```

Dequeue: Delete data in a queue with only one item

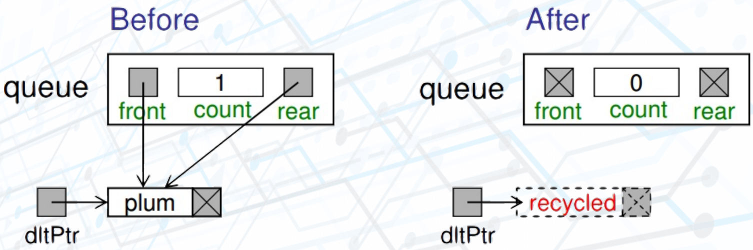


Figure 9: Delete data in a queue with only one item

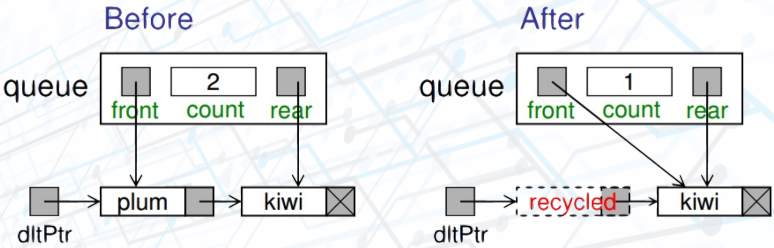


Figure 10: Delete data in a queue with more than one item

Algorithm dequeue(ref queue <metadata>, ref dataOut <dataType>)

Deletes one item at the front of the queue and returns its data to caller

Pre: `queue` is a metadata structure of a valid queue

`dataOut` is to receive dequeued data

Post: front data have been returned to caller

Return `true` if successful, `false` if memory overflow

```
if queue empty then
    return false
end
dataOut = queue.front -> data
dltPtr = queue.front
if queue.count = 1 then
    // Delete data in a queue with only one item
    queue.rear = NULL
end
queue.front = queue.front -> next
queue.count = queue.count - 1
recycle (dltPtr)
return true
End dequeue
```

```
template <class List_ItemType>
int Queue<List_ItemType>::Dequeue(List_ItemType &dataOut){
  if (count == 0)
    return 0;
  dataOut = front->data;
  Node<List_ItemType>* dltPtr= this->front;
  if (count == 1)
    this->rear = NULL;
  this->front = this->front->next;
  this->count--;
  delete dltPtr;
  return 1;
}
```

Queue Front

```
template <class List_ItemType>
int Queue<List_ItemType>::GetQueueFront(List_ItemType &dataOut){
  if (count == 0)
    return 0;
  dataOut = this->front->data;
  return 1;
}
```

Queue Rear

```
template <class List_ItemType>
int Queue<List_ItemType>::GetQueueRear(List_ItemType &dataOut){
  if (count == 0)
    return 0;
  dataOut = this->rear->data;
  return 1;
}
```

Algorithm destroyQueue(ref queue <metadata>)
Deletes all data from a queue

Pre: queue is a metadata structure of a valid queue

Post: queue empty and all nodes recycled

Return nothing

Destroy Queue

```
if queue not empty then
    while queue.front not null do
        temp = queue.front
        queue.front = queue.front->next
        recycle(temp)
    end
end
queue.front = NULL
queue.rear = NULL
queue.count = 0
return
End destroyQueue
```

Destroy Queue

```
template <class List_ItemType>
void Queue<List_ItemType>::Clear() {
  Node<List_ItemType>* temp;
  while (this->front != NULL){
    temp = this->front;
    this->front= this->front->next;
    delete temp;
  }
  this->front = NULL;
  this->rear = NULL;
  this->count = 0;
}
```

Queue Empty

```
template <class List_ItemType>
int Queue<List_ItemType>::IsEmpty() {
  return (this->count == 0);
}

template <class List_ItemType>
int Queue<List_ItemType>::GetSize() {
  return this->count;
}
```

Print Queue

```
template <class List_ItemType>
void Queue<List_ItemType>::Print2Console(){
  Node<List_ItemType>* p;
  p = this->front;
  cout << "Front:␣";
  while (p != NULL){
    cout << p->data << "␣";
    p = p->next;
  }
  cout << endl;
}
```

Using Queue

```
int main(int argc, char* argv[]){
  Queue<int> *myQueue = new Queue<int>();
  int val;
  myQueue->Enqueue(7);
  myQueue->Enqueue(9);
  myQueue->Enqueue(10);
  myQueue->Enqueue(8);
  myQueue->Print2Console();
  myQueue->Dequeue(val);
  myQueue->Print2Console();
  delete myQueue;
  return 1;
}
```

Array-based queue implementation

```
#include <string>
using namespace std;
class ArrayQueue {
private:
  int capacity;
  int front;
  int rear;
  int *storage;

public:
  ArrayQueue(int capacity) {
    storage = new int[capacity];
    this->capacity = capacity;
    front = -1;
    rear = -1;
  }
```

Array-based queue implementation

```
~ArrayQueue() {
  delete[] storage;
}

void enQueue(int value) {
  if(isFull()) throw string("Queue is full");
  if (front == -1) front = 0;
  rear++;
  storage[rear % capacity] = value;
}

void deQueue(int &valueOut) {
  if (isEmpty())
    throw string("Queue is empty");
  valueOut = storage[front % capacity];
  front++;
}
```

```
int getFront() {
  if (isEmpty())
    throw string("Queue is empty");
  return storage[front % capacity];
}

int getRear() {
  if (isEmpty())
    throw string("Queue is empty");
  return storage[rear % capacity];
}
```

Array-based queue implementation

```
bool isEmpty() {
  return (front > rear || front == -1);
}

bool isFull() {
  return (rear - front + 1 == capacity);
}

int getSize() {
  return rear - front + 1;
}

};
```

Using Array-based queue

```
int main(int argc, char* argv[]){
    ArrayQueue *myQueue = new ArrayQueue(10);
    int val;
    myQueue->enQueue(7);
    myQueue->enQueue(9);
    myQueue->enQueue(10);
    myQueue->enQueue(8);
    myQueue->deQueue(val);
    delete myQueue;
    return 1;
}
```

Applications of Queue

Applications of Queue

- Polynomial Arithmetic
- Categorizing Data
- Evaluate a Prefix Expression
- Radix Sort
- Queue Simulation