Traverse LinkedBag.

```
Node<int>* current = bag.head_ptr_;
   while (current != nullptr) {
      std::cout << current->getItem() << " ";
      current = current->getNext();
   }
   std::cout << std::endl;
```

```
#include <iostream>
#include <stdexcept>

class Shape {
public:
   virtual void draw() const = 0; // Pure virtual function
};

class Circle : public Shape {
public:
   void draw() const override {
      std::cout << "Drawing Circle\n";
   }
};

int main() {
   try {
      // Shape s; // Error: Cannot instantiate abstract
class
      Circle c;
      Shape* ptr = &c;
      ptr->draw(); // Calls Circle's draw() due to
polymorphism
   } catch (const std::exception& e) {
      std::cerr << "Exception: " << e.what() << '\n';
   }
   return 0;
}
```

```
void remove(vector<int> &vec, int target) {
   int count = 0;
   for (int i = 0; i < vec.size(); i++)
   {
      vec[count] = vec[i];
      if (vec[i] != target) {
         count++ ;
      }

   }
   vec.resize(count);
}
```

Removing from LinkedBag with preserving the order

```
bool LinkedBag<T>::remove(const T& target) {
 Node<T> prev_ptr = nullptr;
 Node<T> curr_ptr = head_ptr_;
 while (curr_ptr != nullptr && cur_ptr->getItem() != an_entry) {
    prev_ptr  = curr_ptr;
    curr_ptr = curr_ptr->getNext();    }
 if (curr_ptr != nullptr) {
    if (prev_ptr == nullptr) {
     head_ptr_ = curr_ptr->getNext();  }
    else {
     prev_ptr->setNext(curr_ptr->getNext());   }
   delete curr_ptr;
   item_cout--;
   return true;
    }
}
```

```
 template<class T>
 T List<T>::getItem(size_t position) const {
  Node<T>* pos_ptr = getPointerTo(position);
  if(pos_ptr == nullptr)
    throw(std::out_of_range("getItem called with empty list or invalid
 position"));
  else
     return pos_ptr->getItem();   }
```

Table 5 Common Memory Allocation Errors

Statements	Error

`int* p;`
`*p = 5;`
`delete p;`
	There is no call to `new int`.

`int* p = new int;`
`*p = 5;`
`p = new int;`
	The first allocated memory block was never deleted.

`int* p = new int[10];`
`*p = 5;`
`delete p;`
	The `delete[]` operator should have been used.

`int* p = new int[10];`
`int* q = p;`
`q[0] = 5;`
`delete p;`
`delete q;`
	The same memory block was deleted twice.

`int n = 4;`
`int* p = &n;`
`*p = 5;`
`delete p;`
	You can only delete memory blocks that you obtained from calling `new`.

```
// Abstract base class with a pure virtual function
class Shape {
public:
   // Pure virtual function makes Shape an abstract class
   virtual double area() const = 0;
};


// Derived class Circle, inheriting from Shape
class Circle : public Shape {
  public:
   Circle(double r) : radius(r) {}
   // Override the pure virtual function area() in Shape
   double area() const override {
      return 3.14159 * radius * radius;
   }
};

class Rectangle : public Shape {
 private:
   double width, height;
public:
   double area() const override {
      return width * height; }};

int main() {
   // Create instances of Circle and Rectangle
   Circle circle(5.0);
   Rectangle rectangle(4.0, 6.0);
   // Array of Shape pointers to demonstrate polymorphism
   Shape* shapes[] = { &circle, &rectangle };
   // Calculate and print areas using polymorphism
   for (Shape* shape : shapes) {
      std::cout << "Area: " << shape->area() << std::endl;
   }

   return 0;
}
```

```
// polymorphism
 int main()
 {
 string response;
 cout << boolalpha;
 // Make a quiz with two questions
 const int QUIZZES = 2;
 Question* quiz[QUIZZES];
 quiz[0] = new Question;
 quiz[0]->set_text("Who was the inventor of C++?");
 quiz[0]->set_answer("Bjarne Stroustrup");

 ChoiceQuestion* cq_pointer = new ChoiceQuestion;
 cq_pointer->set_text();
 cq_pointer->add_choice("Australia", false);
 quiz[1] = cq_pointer;

 }
```

```
template<class T>
bool LinkedBag<T>::add(const T& new_entry)
{
 // Add to beginning of chain: new node references rest of chain;
 // (head_ptr_ is null if chain is empty)
 Node<T>* new_node_ptr = new Node<T>;
 new_node_ptr->setItem(new_entry);
 new_node_ptr->setNext(head_ptr_); // New node points to
chain
 head_ptr_ = new_node_ptr;// New node is now first node
 item_count_++;
 return true;
} // end add
```

```
class Document {
public:
virtual void convertToPixelArray() const = 0;
virtual int getPriority() const = 0;
private: };
class TextDocument: public Documents  {
public:
//Constructor, destructor
virtual void convertToPixelArray() const override;
virtual int getPriority() const override;
void setFont(const string& font); //text-specific formatting
void setSize(int size);
private:};
Document* myTextDocument = new TextDocument;
Document* myGraphicsDocument = new GraphicsDocument;
//do stuff
myBatchPrinter.addDocument(myTextDocument)
myBatchPrinter.addDocument(myGraphicsDocument)
myBatchPrinter.printAllDocuments();
myTextDocument->convertToPixelArray();
myGraphicsDocument->convertToPixelArray();
```