```
#include "Linked_List.h"

data_t Linked_List::at(int index) const
{
  Node* current = first;

  while (current != 0)
  {
    if (index-- == 0)
    {
      return current->value;
    }
    current = current->next;
  }
  return -1;
}

int Linked_List::index_of(data_t val) const
{
  Node* current = first;
  int index = 0;

  while (current != 0)
  {
    if (current->value == val)
    {
      return index;
    }
    current = current->next;
    index = index + 1;
  }
  return -1;
}
```

```
void Linked_List::push_back(data_t value)
{
  if (last != 0)
  {
    last->next = new Node(value,
                          last->next);
    last = last->next;
  }
  else
  {
    first = new Node(value, first);
    last = first;
  }
}

int Linked_List::size() const
{
  Node* current = first;
  int length = 0;

  while (current != 0)
  {
    current = current->next;
    length = length + 1;
  }
  return length;
}

void Linked_List::clear()
{
  delete first; // Recursion!
  first = last = 0;
}
```

```
data_t Linked_List::remove_at(int index)
{
  Node* previous = 0;
  Node* current = first;

  while (current != 0 && index--)
  {
    previous = current;
    current = current->next;
  }
  Node* victim = current;

  if (victim == 0) // non-existing index?
  {
    return -1;
  }

  if (previous != 0)
  {
    previous->next = victim->next;
  }
  if (victim == first)
  {
    first = first->next;
  }
  if (victim == last)
  {
    last = previous;
  }

  data_t value = victim->value;
  victim->next = 0;
  delete victim;

  return value;
}
```

```
void Linked_List::insert_at(int index,
                            data_t value)
{
  Node* prev = 0;
  Node* current = first;

  while (current != 0 && index-- > 0)
  {
    prev = current;
    current = current->next;
  }

  if (prev == 0) // first
  {
    first = new Node(value, first);

    if (last == 0) // was it empty ?
    {
      last = first;
    }
  }
  else // inner element
  {
    prev->next = new Node(value,
                          prev->next);
    if (current == 0) // was it last ?
    {
      last = last->next;
    }
  }
}
```

```
Linked_List::Node*
Linked_List::clone(Node* src)
{
  if (src == 0)
    return 0;
  else
    return new Node(src->value,
                    clone(src->next));
}

Linked_List::
Linked_List(Linked_List const& rhs)
{
  first = clone(rhs.first);

  Node* current = first;

  while (current != 0)
  {
    last = current;
    current = current->next;
  }
}
```

```
Linked_List
Linked_List::
operator=(Linked_List const& rhs)
{
  if (this != &rhs)
  {
    Linked_List copy(rhs);

    Node* save_slot = first;
    first = copy.first;
    // Will deleted by destructor for copy
    copy.first = save_slot;

    last = copy.last;
    copy.last = 0; // Only good habit
  }
  return *this;
}

std::ostream&
operator<<(std::ostream& os,
           Linked_List const& l)
{
  Linked_List::Node* current = l.first;

  while (current != 0)
  {
    os << current->value << std::endl;
    current = current->next;
  }
  return os;
}
```