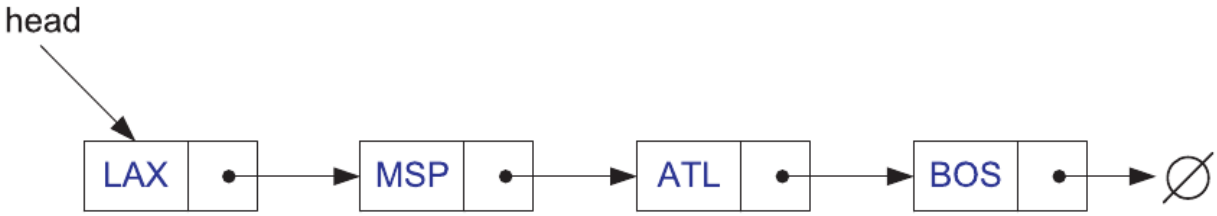
Danh sách liên kết đôi

Danh sách liên kết đơn vs. danh sách liên kết đôi

Đối với danh sách liên kết đơn, thao tác xóa phần tử ở đầu danh sách có thể được cài đặt khá đơn giản. Tuy nhiên, thao tác xóa phần tử ở cuối khá phức tạp. Ta cùng xem làm thế nào để xóa phần tử ở cuối danh sách.



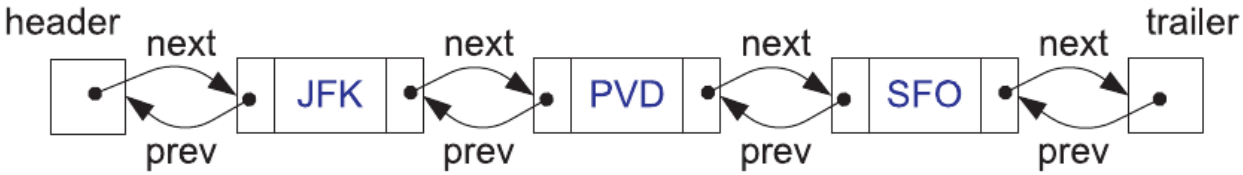
Để xóa nút với phần tử **BOS** ở cuối danh sách, ta phải duyệt danh sách từ đầu tới cuối, tìm được nút trỏ tới nút cuối này (tức là nút với phần tử **ALT**), xóa nút cuối và đặt con trỏ trỏ đến nút cuối bằng **NULL**. Độ phức tạp của thao tác này là $O(n)$ với n là kích thước của danh sách.

Thao tác xóa phần tử ở cuối danh sách có thể được cài đặt như dưới đây. Liệu bạn có cách để cài đặt ngắn gọn hơn?

```
template<typename E>
void SLinkedList<E>::removeBack(){
    if (empty()) return;        // Nếu danh sách rỗng
    if (head->next == NULL)           // Nếu danh sách chỉ có 1 phần tử
        delete head;
        head = NULL;
        return ;
    }                                 // Danh sách có >= 2 nút
    SNode<E> *v1 = head;              // Trỏ đến nút đầu
    SNode<E> *v2 = head->next;        // Trỏ đến nút thứ 2
    while (v2->next != NULL) {        // Nút v2 chưa là nút cuối cùng
        v1 = v2;                      // v1 là nút trước nút v2
        v2 = v2->next;
    }
    delete v2;                       // Xóa nút cuối là nút v2
    v1->next = NULL;                 // để v1 trở thành nút cuối
}
```

Khó khăn của thao tác xóa phần tử cuối đến từ sự kiện rằng: **trong danh sách liên kết đơn, ta chỉ di chuyển theo một hướng bằng con trỏ** `next` . Ta không có cách nào truy cập đến nút trước của một nút. Để vượt qua khó khăn này, ta sử dụng **danh sách liên kết đôi**. Danh sách này cho phép di chuyển giữa các nút theo cả hai hướng. Tên gọi danh sách liên kết đôi bởi vì mỗi nút có hai con trỏ, `next` và `prev`, tương ứng trỏ đến nút trước và nút sau của mỗi nút.

Hình dưới đây là một ví dụ về danh sách liên kết đôi.



Trong danh sách liên kết đôi, ta dùng hai con trỏ `header` và `trailer` để trỏ đến nút đầu và nút cuối danh sách. Nhờ đó, ta có thể di chuyển theo cả hai hướng: từ đầu về cuối và từ cuối về đầu. Hơn nữa, ở tại mỗi nút, ta có thể di chuyển về nút phía sau qua con trỏ `next` hoặc về nút phía trước quả con trỏ `prev`.

Cài đặt danh sách liên kết đôi

Các nút của danh sách liên kết đôi được cài đặt như sau.

```
template<typename > class DLinkedList;
template<typename E>
class DNode
{
 private:
  E elem;
  DNode<E> *prev;
  DNode<E> *next;
  friend class DLinkedList<E>;
};
```

Để thuận tiên khi cài đặt, trong lớp `DLinkedList` ta thêm hai phần tử cầm canh `header` và `trailer`. Khác với `head` trong danh sách liên kết đơn (chỉ là con trỏ), `header` và `trailer` là **hai nút thực sự** đứng đầu và đứng cuối danh sách. Lớp `DLinkedList` có giao diện như sau:

```
template <typename E>
class DLinkedList
{
 private:
  DNode<E> *header;
  DNode<E> *trailer;
 public:
  DLinkedList();
  ~DLinkedList();
  bool empty();
  const  E & front() const ;
  const  E & back() const ;
  void addFront(const E &e);
  void addBack(const E &e);
  void removeFront();
  void removeBack();
```

```
  void print() const;
 protected:                          // Sử dụng cục bộ trong lớp hoặc lớp con
  void add (DNode<E> *v, const E &e);      // Thêm một nút trước nút v
  void remove (DNode<E> *v);                 // Xóa nút v
};
```

Ngoài các hàm cấu tử, hủy tử, `empty`, `addFront`, `addBack`, `removeFront`, `removeBack`, và `print` là quen thuộc. Ta còn hai hàm `add` và `remove`:

- `void add (DNode<E> *v, const E &e` để thêm phần tử `e` vào ngay trước nút trỏ bởi `v`;
- `void remove (DNode<E> *v)` để xóa nút trỏ bởi `v` khỏi danh sách.

Hai hàm này là hàm thành viên `protected`. Vì vậy, nó chỉ được truy cập trong lớp và các lớp con. Nó không được gọi bên ngoài lớp.

Cài đặt cấu tử và hủy tử

Điểm khác biệt so với danh sách liên kết đơn là ta phải cấp phát hai nút `header` và `trailer`. Nút đầu tiên trong danh sách được trỏ bởi `header->next`, nút cuối cùng trong danh sách được trỏ bởi `trailer->prev`. Phần dữ liệu của `header` và `trailer` không dùng.

```
template <typename E>
DLinkedList<E>::DLinkedList() {      // Cấu tử
    header = new DNode<E>;           // tạo ra phần tử cầm canh
    trailer = new DNode<E>;
    header->next = trailer;          // trỏ tới nhau
    trailer->prev = header;
}

template <typename E>
DLinkedList<E>::~DLinkedList(){
  while (!empty()) removeFront();
  delete header;
  delete trailer;
}
```

Các hàm `empty`, `front`, và `back`

Các hàm này được cài đặt một cách đơn giản tự nhiên. Ta nhấn mạnh lại rằng: Nút đầu tiên trong danh sách được trỏ bởi `header->next`, nút cuối cùng trong danh sách được trỏ bởi `trailer->prev`.

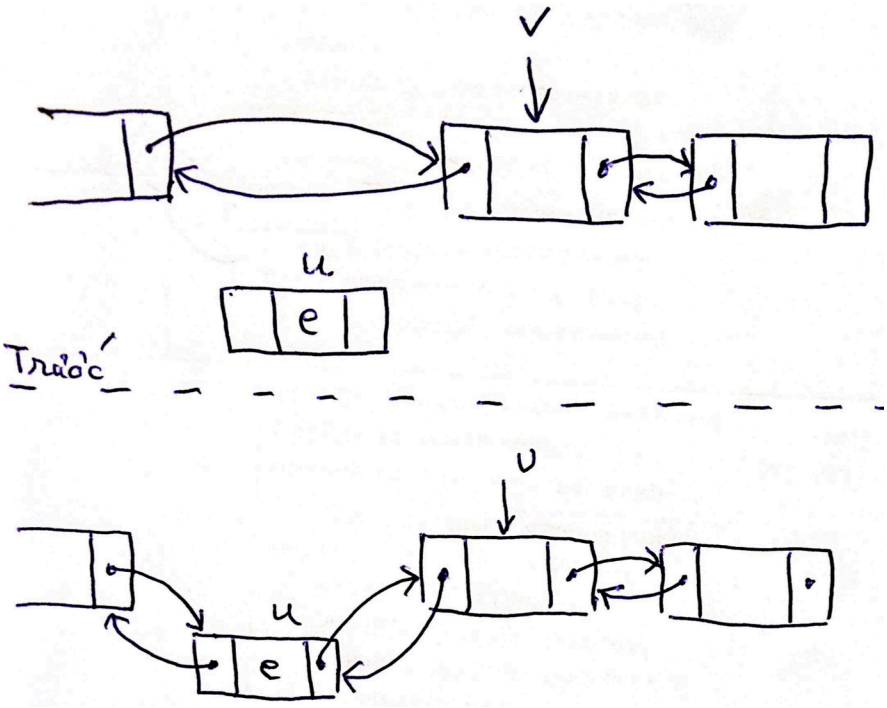
```
template <typename E>
bool DLinkedList<E>::empty ()
{ return header->next == trailer; }

template <typename E>
const E & DLinkedList<E>::front() const
{ return header->next->elem;  }

template <typename E>
const E & DLinkedList<E>::back() const
{ return trailer->prev->elem; }
```

Thêm nút vào danh sách

Thao tác thêm một phần tử `e` vào trước nút trỏ bởi `v` được mô tả bởi hình sau:



và được cài đặt như sau:

```
//Thêm một phần tử e vào trước  nút v
template<typename E>
void DLinkedList<E>::add(DNode<E>* v, const E& e) {
  DNode<E>* u = new DNode<E>;
  u->elem = e;          // tạo một nút mới cho  e
  u->next = v;          // nối u với v
  u->prev = v->prev;
  v->prev->next = v->prev = u;
}
```

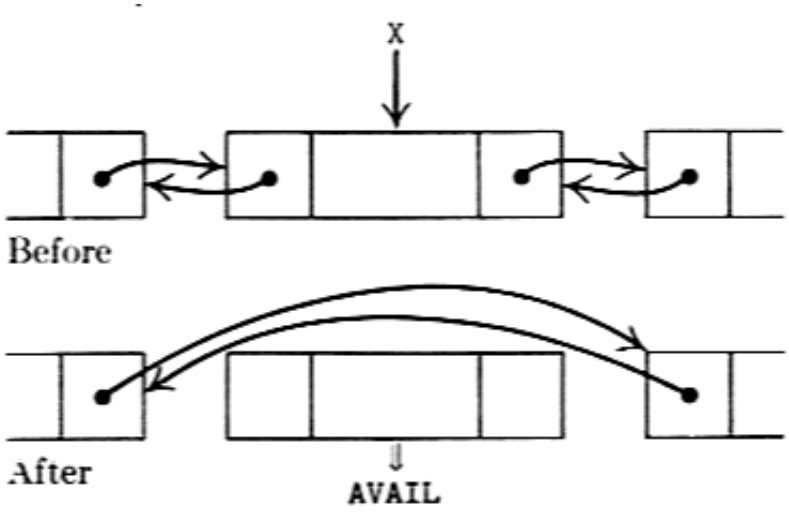
Nhờ có hàm này, các hàm `addFront` và `addBack` được cài đặt một cách đơn giản:

```
//Thêm phần tử vào đầu danh sách
template<typename E>
void DLinkedList<E>::addFront(const E& e)
{ add(header->next, e); }

// Thêm phần tử vào cuối danh sách
template<typename E>
void DLinkedList<E>::addBack(const E &e)
{ add (trailer->prev, e); }
```

Xóa một nút

Thao tác xóa nút trỏ bởi `v` được mô tả như hình sau:



và được cài đặt như dưới đây.

```
template <typename E>
void DLinkedList<E>::remove (DNode<E> *v)  // Xóa nút v
{
  v->prev->next = v->next;
  v->next ->prev = v->prev;
  delete v;
}
```

Nhờ có hàm này, các hàm `removeFront` và `removeBack` được cài đặt rất đơn giản:

```
// Xóa phần tử ở đầu danh sách
template <typename E>
void DLinkedList<E>::removeFront()
{ remove(header->next); }

// Xóa phần tử ở cuối danh sách
template <typename E>
void DLinkedList<E>::removeBack()
{ remove(trailer->prev); }
```

In danh sách và hàm `main`

Để in danh sách, ta duyệt từ nút đầu tiên `header->next` cho đến trước khi gặp nút `trailer`.

```
template <typename E>
void DLinkedList<E>::print() const {
  DNode<E> * v = header->next;
  cout<<"header <-> ";
  while (v != trailer) {
    cout<<v->elem<<" <-> ";
    v = v->next;
  }
  cout<<" trailer"<<endl;
}
```

Dưới đây là hàm `main` để test chương trình.

```
int main(int argc, char const *argv[]) {
    DLinkedList<int> x;
    x.addFront(1);  x.addFront(3);
    x.addFront(5);  x.addFront(7);
    x.addFront(9);
    x.print();
}
```

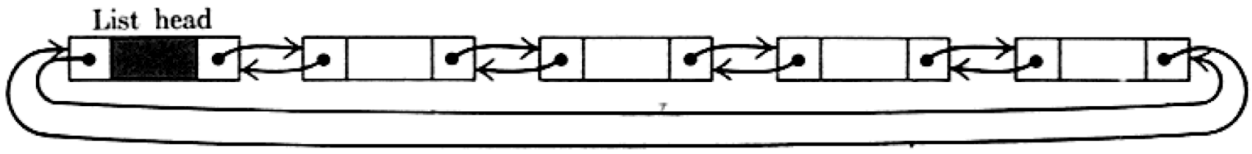
Bài tập

Bài tập 1: Cài đặt thêm các hàm sau cho danh sách liên kết đôi:

- `get(i)` trả về giá trị tại chỉ số `i`. *Mở rộng:* hãy cài đặt với toán tử `[]`.
- `add(i,x)` chèn giá trị `x` sau chỉ số `i` . Các giá trị sau chỉ số `i` sẽ có chỉ số tăng lên 1.
- `remove(i)` xóa giá trị tại chỉ số `i`. Các giá trị có chỉ số sau `i` sẽ nhận giá trị giảm đi 1.

Bài tập 2: Hãy cài đặt lớp `Vector` sử dụng danh sách liên kết đôi. *Yêu cầu:* Bạn phải in mã nguồn và nộp trên lớp trong buổi học tới.

Bài tập 3: Ta có thể cài đặt danh sách liên kết đôi mà không có nút `trailer`. Khi đó, nút `prev` của nút `header` sẽ trỏ tới nút cuối; còn `next` của nút cuối sẽ trỏ về nút `header` như hình dưới đây. Nếu danh sách rỗng thì ta phải có `header->next = header->prev = header`.



Bạn hãy cài đặt danh sách liên kết đôi như mô tả ở trên.

Lớp `list` trong thư viện chuẩn `std`

Thư viện chuẩn `std` cung cấp cho ta một cài đặt danh sách liên kết đôi gọi là `list`.

Ví dụ dưới đây dùng `list` để cài đặt các thao tác với danh bạ điện thoại. Do thao tác thêm và xóa phần tử trong danh bạ điện thoại xuất hiện khá thường xuyên, nên việc sử dụng cấu trúc `list` ở đây rất thích hợp.

```
#include <list>
using namespace std;
// Cấu trúc dữ liệu cho một số điện thoại (tên người, số điện thoại)
struct Entry {
  string name;
  int number;
};
// Danh bạ điện thoại là một danh sách các số điện thoại
list<Entry> phone_book = {
    {"David Hume",123456},
  {"Karl Popper",234567},
  {"Bertrand Ar thur William Russell",345678}
};
```

Khi sử dụng `list`, ta không truy cập phần tử dùng chỉ số `[]` như mảng hoặc `vector`. Thay vào ta có thể tìm kiếm một phần tử trong `list` với một giá trị cho trước.

```
int get_number(const string& s)
{
  for (const auto& x : phone_book)
    if (x.name==s)  return x.number;
  return 0; // dùng 0 để biểu diễn  "không thấy số"
}
```

Việc tìm phần tử `s` được thực hiện từ đầu danh sách cho đến khi tìm thấy `s` hoặc tới cuối của `phone_book`.

Đôi khi, ta cần xác định một nút của `list`. Ví dụ, ta muốn xóa nó hoặc thêm một nút mới trước nó. Để làm điều này ta sử dụng `iterator`: một `list iterator` xác định một nút của `list` và có thể được dùng để lặp qua một `list` (dùng tên của nó).

Bạn có thể đọc về `iterator` tại link https://cpp.daynhauhoc.com/11/2-stl-iterators/.

Mọi thư viện chuẩn `containter` đều cung cấp các hàm `begin()` và `end()`, tương ứng, để trả về một `iterator` tới nút đầu và nút cuối của danh sách. Bằng cách dùng tường minh `iterator`, ta có thể viết lại hàm `get_number()` như sau:

```
int get_number(const string& s)
{
  for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
    if (p->name==s)   return p->number;
  return 0; // dùng 0 để biểu diễn  "không thấy số"
}
```

Đưa ra một `iterator p`, `*p` là nút mà nó trỏ tới, `++p` di chuyển nút `p` tới nút tiếp theo, tức là `p=p->next`, và khi `p` trỏ tới một lớp, vậy thì `p->m` tương đương với `(*p).m`.

Ta có thể dễ dàng thêm hoặc xóa phần tử trong `list`.

```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
  phone_book.insert(p, ee);  // thêm ee trước nút trỏ bởi p
  phone_book.erase(q);       // xóa nút trỏ bởi q
}
```

Chú ý: Khi chúng ta muốn xử lý dãy, ta có thể chọn giữa dùng `vector` hoặc `list`. Trừ khi bạn có lý do nào đó để sử dụng `list`, nếu không hãy dùng `vector` trong mọi trường hợp. Vector thực hiện tốt hơn cho việc duyệt (ví dụ `find()` và `count()`) và cho việc sắp xếp và tìm kiếm (cụ thể, `sort()` và `binary_search()`).