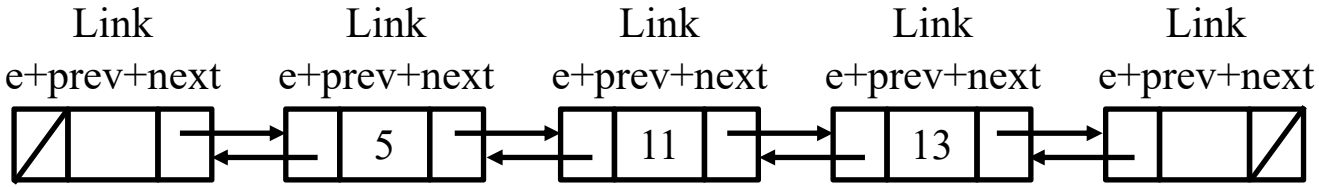
List

structure abstraction

- **Linked list implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new list elements as needed
  - *singly linked list*
  - *doubly linked list*
    - doubly *linked node* (class *Link*)

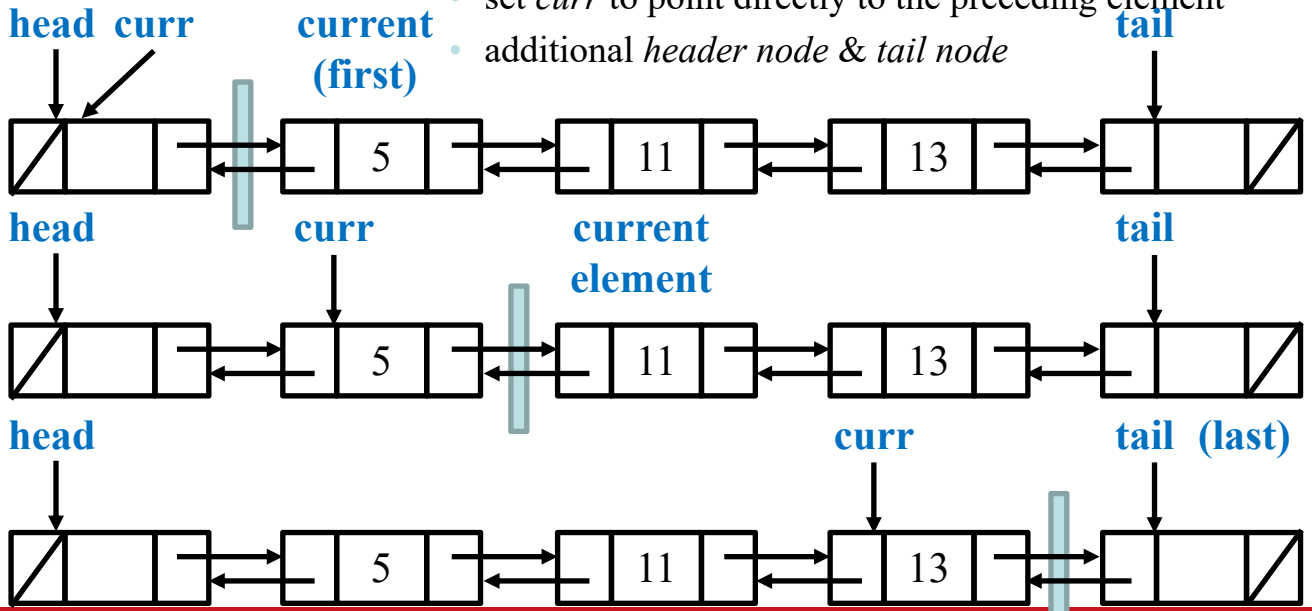


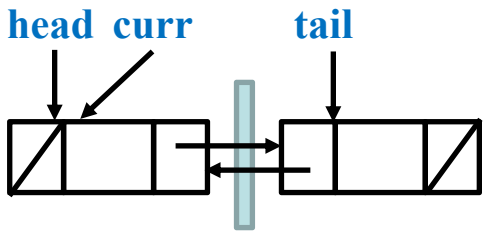

```
template <class T> class Link{ // doubly linked list (primitive version)
public: T e; // element
        Link *next; // pointer to next link node in list
        Link *prev; // pointer to previous link node
        Link(const T& ei,Link* previ=NULL,Link* nexti=NULL){
                e=ei;next=nexti;prev=previ;}
        Link(Link* previ=NULL,Link* nexti=NULL){next=nexti;prev=previ;}
};
```

List

structure abstraction

- **Linked list implementation**
  - *doubly linked list*
    - doubly *linked node* (efficient *prev* as well as *next*)
    - set *curr* to point directly to the preceding element
    - additional *header node* & *tail node*



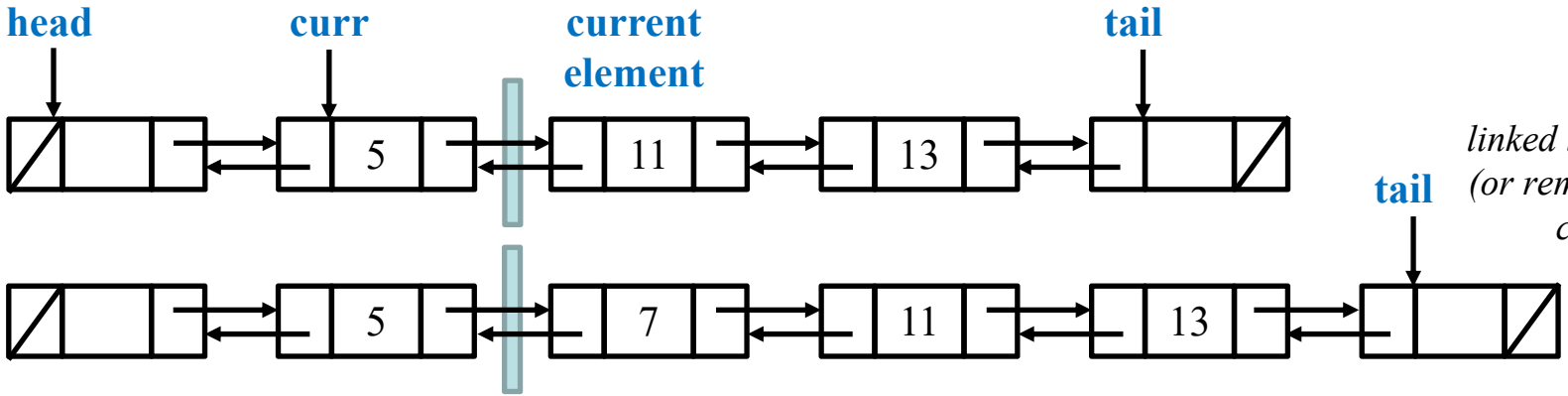


initial state of a doubly linked list when using a header node & a tail node

List

structure abstraction

- **Linked list implementation**
  - *doubly linked list*
    - doubly *linked node* (efficient *prev* as well as *next*)
    - set *curr* to point directly to the preceding element
    - additional *header node* & *tail node*
  - *efficient insert & remove*
    - pointer adjustment only, no tedious shifting of elements

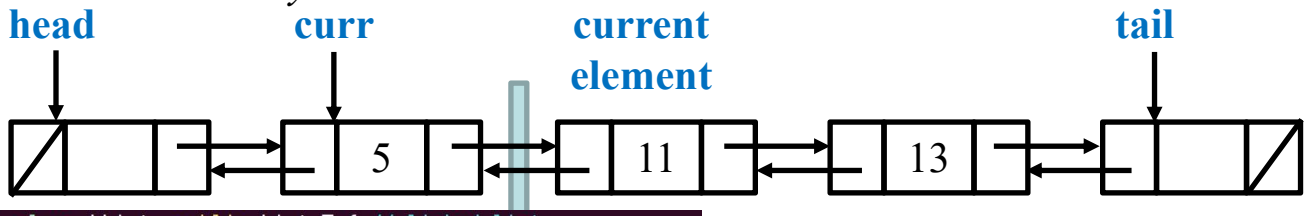


linked list after insertion (or removal) of the node containing 7

List

- **Linked list implementation**
  - *doubly linked list*

structure abstraction

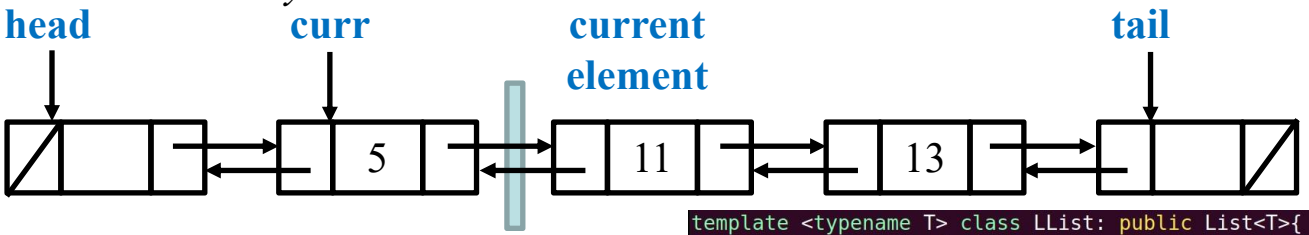


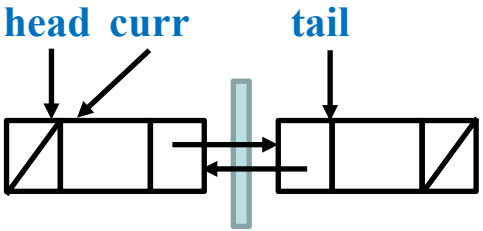

```
template <typename T> class LList: public List<T>{ // linked list
private:Link<T> *head,*tail,*curr; // pointers to list header,last,current
        int n; // list length
        void init(){curr=head=new Link<T>;tail=new Link<T>;n=0;
                head->next=tail;tail->prev=head;}
        void removeall(){ // return link nodes to free store
                while(head!=NULL){curr=head;head=head->next;delete curr;}}
public: LList(){init();} ~LList(){removeall();}
        int length() const{return n;}
        int currP() const{Link<T>* tmp=head; // no direct indexing
                int i;for(i=0;tmp!=curr;i++) tmp=tmp->next; return i;}
        const T& getE() const{assert(curr->next!=tail);return curr->next->e;}
                // curr is preceding, so curr->next is current
        void prev(){if(curr!=head) curr=curr->prev;} // convenient than SLList
        void next(){if(curr!=tail->prev) curr=curr->next;}
        void moveToPos(int pos){ // position is {0,1,2,...,n-1,n}
                assert(pos>=0 && pos<=n);curr=head;
                for(int i=0;i<pos;i++) curr=curr->next;}
        void moveToStart(){curr=head;}
        void moveToEnd(){curr=tail->prev;} // last is the one before tail
        void insert(const T& it){curr->next=curr->next->prev=
                new Link<T>(it,curr,curr->next);n++;} // Reflect why
        void append(const T& it){tail->prev=tail->prev->next=
                new Link<T>(it,tail->prev,tail);n++;} // Reflect why
        T remove(){assert(curr->next!=tail); T it=curr->next->e;
                Link<T>* tmp=curr->next;curr->next->next->prev=curr;
                curr->next=curr->next->next;delete tmp;n--;return it;}
        void clear(){removeall();init();}
        void S(){Link<T>* t=head; while(t->next!=curr->next){t=t->next;
                std::cout<<t->e<<' ';} std::cout<<"| "; while(t->next!=tail){
                t=t->next;std::cout<<t->e<<' ';} std::cout<<'\n';}
};
```

List

structure abstraction

- **Linked list implementation**
  - *doubly linked list*





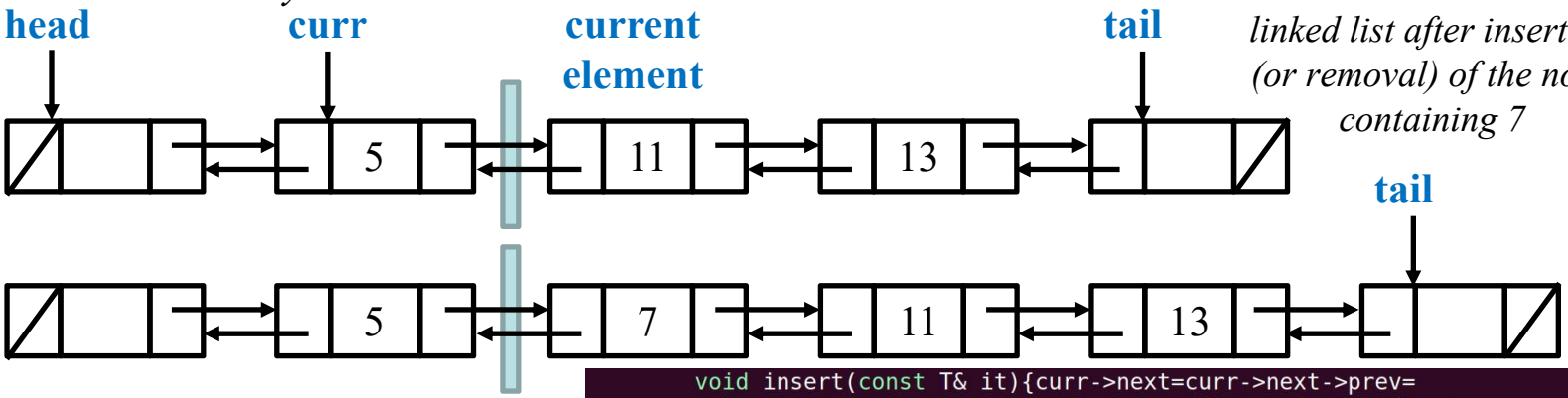
initial state of a doubly linked list when using a header node & a tail node

```
template <typename T> class LList: public List<T>{ // linked list
private:Link<T> *head,*tail,*curr; // pointers to list header,last,current
        int n; // list length
        void init(){curr=head=new Link<T>;tail=new Link<T>;n=0;
                head->next=tail;tail->prev=head;}
        void removeall(){ // return link nodes to free store
                while(head!=NULL){curr=head;head=head->next;delete curr;}}
public: LList(){init();} ~LList(){removeall();}
        int length() const{return n;}
        int currP() const{Link<T>* tmp=head; // no direct indexing
                int i;for(i=0;tmp!=curr;i++) tmp=tmp->next; return i;}
        const T& getE() const{assert(curr->next!=tail);return curr->next->e;}
                // curr is preceding, so curr->next is current
        void prev(){if(curr!=head) curr=curr->prev;} // convenient than SLList
        void next(){if(curr!=tail->prev) curr=curr->next;}
        void moveToPos(int pos){ // position is {0,1,2,...,n-1,n}
                assert(pos>=0 && pos<=n);curr=head;
                for(int i=0;i<pos;i++) curr=curr->next;}
        void moveToStart(){curr=head;}
        void moveToEnd(){curr=tail->prev;} // last is the one before tail
```

List

structure abstraction

- **Linked list implementation**
  - *doubly linked list*




```
        void insert(const T& it){curr->next=curr->next->prev=
                new Link<T>(it,curr,curr->next);n++;} // Reflect why
        void append(const T& it){tail->prev=tail->prev->next=
                new Link<T>(it,tail->prev,tail);n++;} // Reflect why
        T remove(){assert(curr->next!=tail); T it=curr->next->e;
                Link<T>* tmp=curr->next;curr->next->next->prev=curr;
                curr->next=curr->next->next;delete tmp;n--;return it;}
        void clear(){removeall();init();}
        void S(){Link<T>* t=head; while(t->next!=curr->next){t=t->next;
                std::cout<<t->e<<' ';} std::cout<<"| "; while(t->next!=tail){
                t=t->next;std::cout<<t->e<<' ';} std::cout<<'\n';}
};
```

List

- **Linked list implementation**
  - *doubly linked list*

usage of the *linked list* is the same
be it a *singly linked list* or *doubly linked list*

```
g++ demoLList.cpp -o _a; ./_a;
| 5 4 3 2 1
| 6 5 4 3 2 1
6 5 4 3 2 1 |
6 5 4 3 2 1 | 7
6 5 4 3 2 1 | 8 7
6 5 4 | 3 2 1 8 7
6 5 4 | -1 3 2 1 8 7
6 5 4 | -2 -1 3 2 1 8 7
6 5 4 -2 -1 | 3 2 1 8 7
3 removed! 6 5 4 -2 -1 | 2 1 8 7
2 removed! 6 5 4 -2 -1 | 1 8 7
| 6 5 4 -2 -1 1 8 7
6 removed! | 5 4 -2 -1 1 8 7
List currently has 7 elements
| one two
one | two
one | three two
one | four three two
one four | three two
one four | two
three is just removed from list
one four |
two is just removed from list
```

```
#include <iostream>
#include "LList.h"
using namespace std;

template <class T> void ShowL(LList<T>& a){a.S();}

int main(){
        LList<int> ai; for(int i=5;i>0;i--) ai.append(i); ai.S();ai.insert(6);ai.S();
        ai.moveToEnd();ai.S();ai.insert(7);ai.S();ai.insert(8);ai.S();
        ai.moveToPos(3);ai.S();ai.insert(-1);ai.S();ai.insert(-2);ai.S();
        ai.moveToPos(5);ShowL<int>(ai);
        int e=ai.remove();cout<<e<<" removed! ";ai.S();
        e=ai.remove();cout<<e<<" removed! ";ai.S(); ai.moveToStart();ai.S();
        e=ai.remove();cout<<e<<" removed! ";ai.S();
        cout<<"List currently has "<<ai.length()<<" elements\n";
        // ai.moveToEnd();ai.S();ai.remove();

        LList<const char*> as;as.append("one");as.append("two");as.S();
        as.moveToPos(1);as.S();as.insert("three");as.S();as.insert("four");as.S();
        as.moveToPos(2);ShowL<const char*>(as);
        const char* ec=as.remove();as.S();cout<<ec<<" is just removed from list\n";
        ec=as.remove();as.S();cout<<ec<<" is just removed from list\n"; return 0;
}
```

List

- **free list**
  - take advantage of already allocated space

doubly linked list with freelist
overloaded new via individual ::new

overloading of *new* is the same
be it *singly linked* or *doubly linked*

```
template <class T> class Link{ // doubly linked list with freelist
private:static Link<T>* fL; // all Link objects share the pointer 'fL'
public: T e; // element
        Link *next; // pointer to next link node in list
        Link *prev; // pointer to previous link node
        Link(const T& ei,Link* previ=NULL,Link* nexti=NULL){
                e=ei;next=nexti;prev=previ;}
        Link(Link* previ=NULL,Link* nexti=NULL){next=nexti;prev=previ;}
        void* operator new(size_t){ // 'new' operator overloading
                if (NULL==fL) return ::new Link; // create space
                // ::new is the standard C++ system call
                Link<T>* tmp=fL;fL=fL->next;return tmp;} // reuse freelist
        void operator delete(void* p){ // 'delete' operator overloading
                ((Link<T>*)p)->next=fL;fL=(Link<T>*)p;} // add to freelist
};
template <class T> Link<T>* Link<T>::fL=NULL; // create freelist head
```

List

- **free list**
  - take advantage of already allocated space

```
template <class T> class Link{ // doubly linked list with freelist
private:static Link<T>* fL; // all Link objects share the pointer 'fL'
        const static int fN=100; // number of batch 'new' for freelist
public: T e;Link *next,*prev; // element, pointers to next & previous link nodes
        Link(const T& ei,Link* previ=NULL,Link* nexti=NULL){
                e=ei;next=nexti;prev=previ;}
        Link(Link* previ=NULL,Link* nexti=NULL){next=nexti;prev=previ;}
        void* operator new(size_t){ // 'new' operator overloading
                if (NULL==fL){ // create space in batch if freelist is empty
                Link<T>* t=::new Link<T>[fN]; t[fN-1].next=NULL;
                for(int i=fN-2;i>=0;i--) t[i].next=&t[i+1]; // linking
                fL=&t[1];return t;} // add last fN-1 ones to freelist
                Link<T>* tmp=fL;fL=fL->next;return tmp;} // reuse freelist
// If freelist is empty & overloaded 'new Link<T>' is invoked, space of fN Link objects
// will be allocated with their 'next' set in above 'linking'. t[0].next is set as well.
// Since overloaded 'new' returns 'void*', which is converted to 'Link<T>*' whose pointed
// Link object space i.e. t[0] is made by constructor II and has 'next=NULL' by default.
// So among the fN Link objects, the first i.e. t[0] desirably has 'next' reset to NULL,
// whereas t[1:fN-1] that are added to freelist desirably keep 'next' set in 'linking'.
// Similar logic applies when overloaded 'new Link<T>(const T&,Link*)' is invoked.
        void operator delete(void* p){ // 'delete' operator overloading
                ((Link<T>*)p)->next=fL;fL=(Link<T>*)p;} // add to freelist
};
template <class T> Link<T>* Link<T>::fL=NULL; // create freelist head
```

doubly linked list with freelist
overloaded new via batch ::new

overloading of *new* is the same
be it *singly linked* or *doubly linked*

Stack

- **Generic list => generic stack (last-in-first-out i.e. LIFO)**
  - a **finite**, **ordered** sequence of data items known as *elements*
    - *ordered* means each element has a *position* in the list, namely a list is a set of elements with *ordinal numbers*
  - *length* (empty if 0)
  - *position* ~~*& current position*~~ => *tail*
  - *get* (~~current element~~) => *tail*
  - ~~*head &*~~ *tail*
  - *next* & *prev*(ious)
  - ~~*move* (to start, end, or any position)~~
  - ~~*insert &*~~ *append* (=> *tail*)
  - *remove* => *tail*
  - *clear*

structure abstraction

Stack

- **Generic list => generic stack (last-in-first-out i.e. LIFO)**
  - a **finite**, **ordered** sequence of data items known as *elements*
    - *ordered* means each element has a *position* in the list, namely a list is a set of elements with *ordinal numbers*
  - *length* (empty if 0)
  - *position* & *get* & *tail* => ***top***
  - *append* & *next* & *tail* => ***push***
  - *remove* & *prev* & *tail* => ***pop***
  - *clear*

structure abstraction

Stack

- Generic stack (last-in-first-out i.e. LIFO)

structure abstraction

```
#ifndef __STACK_H__
#define __STACK_H__
template <typename T> class Stack{ // Stack template (ADT)
private:void operator=(const Stack&){} // protect assignment (reflect why)
        Stack(const Stack&){} // protect copy constructor (reflect why)
public: Stack(){} // default constructor
        virtual ~Stack(){} // base destructor
        virtual int length() const=0; // return the number of stack elements
        virtual const T& top() const=0; // copy of the top element
        virtual void push(const T& item)=0; // push onto stack top
        virtual T pop()=0; // remove & return the top element
        virtual void clear()=0; // make stack empty
        virtual void S()=0; // show stack elements
};
#endif
```

Stack

- **Array-based stack implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - *length* (empty if 0)
  - *position* & *get* & *tail* => ***top***
  - *append* & *next* & *tail* => ***push***
  - *remove* & *prev* & *tail* => ***pop***
  - *clear*

Stack

- **Array-based stack implementation**

structure abstraction

```
#ifndef __ASTACK_H__
#define __ASTACK_H__
#include <iostream>
#include <assert.h>
#include "Stack.h"

#define STACK_DEFAULT_MAX_N 1000
template <typename T> class AStack: public Stack<T>{ // array-based stack
private:int maxN; // maximum allowable number of stack
        int t; // index of top element
        T* e; // array holding stack elements
public: AStack(int ni=STACK_DEFAULT_MAX_N){maxN=ni;t=0;e=new T[maxN];}
        ~AStack(){delete[] e;} // destructor: deallocate array space
        int length() const{return t;}
        const T& top() const{assert(t!=0 && "Stack is empty");
                return e[t-1];} // copy of the top element
        void push(const T& item){assert(t!=maxN && "Stack is full");
                e[t++]=item;} // push onto stack top
        T pop(){assert(t!=0 && "Stack is empty");
                return e[--t];} // remove & return the top element
        void clear(){t=0;} // re-initialize
        void S(){for(int i=0;i<t;)std::cout<<e[i++]<<' ';std::cout<<"|TOP\n";}
};
#endif
```

Stack

- **Array-based stack implementation**

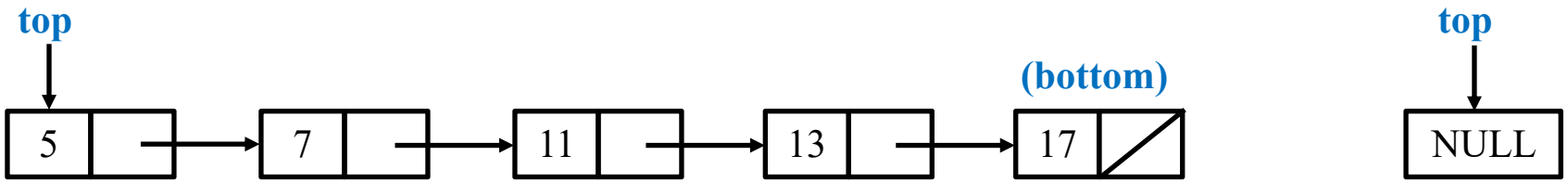
structure abstraction

```
 g++ demoAStack.cpp -o _a; ./_a
Empty stack: |TOP
Push 1 into stack: 1 |TOP
Push 2 into stack: 1 2 |TOP
Push 3 into stack: 1 2 3 |TOP
Push 4 into stack: 1 2 3 4 |TOP
Stack currently has 4 elements
Pop 4 from stack: 1 2 3 |TOP
Pop 3 from stack: 1 2 |TOP
Get 2 from stack: 1 2 |TOP
Stack currently has 2 elements
Push 13 into stack: 1 2 13 |TOP
Stack currently has 3 elements
Empty stack: |TOP
Push one into stack: one |TOP
Push two into stack: one two |TOP
Push three into stack: one two three |TOP
Stack currently has 3 elements
Pop three from stack: one two |TOP
Get two from stack: one two |TOP
Stack currently has 2 elements
Push algorithms into stack: one two algorithms |TOP
Stack currently has 3 elements
```

```
#include <iostream>
#include "AStack.h"
using namespace std;
int main(){AStack<int> ai;cout<<"Empty stack: ";ai.S();for(int i=1;i<=4;i++){
                cout<<"Push "<<i<<" into stack: ";ai.push(i);ai.S();}
        cout<<"Stack currently has "<<ai.length()<<" elements\n";
        int e=ai.pop();cout<<"Pop "<<e<<" from stack: ";ai.S();
        e=ai.pop();cout<<"Pop "<<e<<" from stack: ";ai.S();
        e=ai.top();cout<<"Get "<<e<<" from stack: ";ai.S();
        cout<<"Stack currently has "<<ai.length()<<" elements\n";
        cout<<"Push "<<13<<" into stack: ";ai.push(13);ai.S();
        cout<<"Stack currently has "<<ai.length()<<" elements\n";

        AStack<const char*> as;cout<<"Empty stack: ";as.S();
        cout<<"Push one into stack: ";as.push("one");as.S();
        cout<<"Push two into stack: ";as.push("two");as.S();
        cout<<"Push three into stack: ";as.push("three");as.S();
        cout<<"Stack currently has "<<as.length()<<" elements\n";
        const char* ec=as.pop();cout<<"Pop "<<ec<<" from stack: ";as.S();
        ec=as.top();cout<<"Get "<<ec<<" from stack: ";as.S();
        cout<<"Stack currently has "<<as.length()<<" elements\n";
        cout<<"Push algorithms into stack: ";as.push("algorithms");as.S();
        cout<<"Stack currently has "<<as.length()<<" elements\n";return 0;
}
```

Stack

- **Linked stack implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new stack elements as needed
  - *singly linked stack* (already effecient enough)

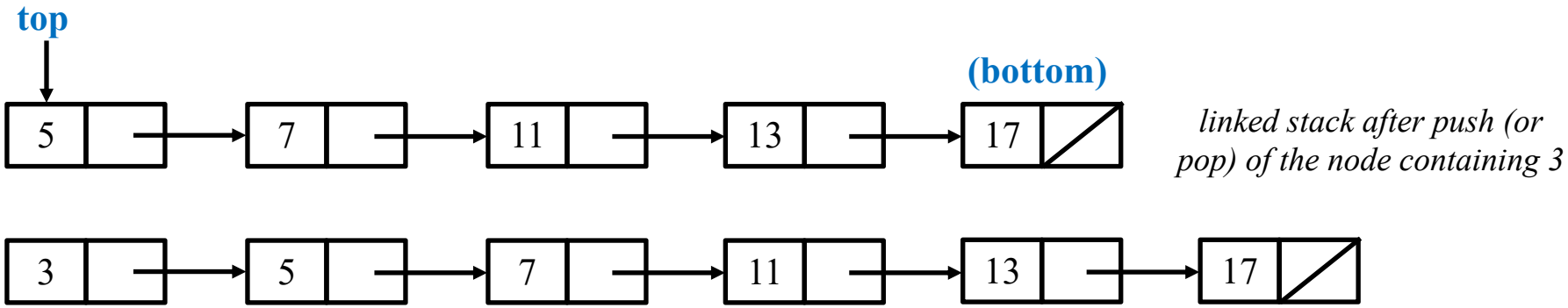


initial state of a linked stack

Stack

structure abstraction

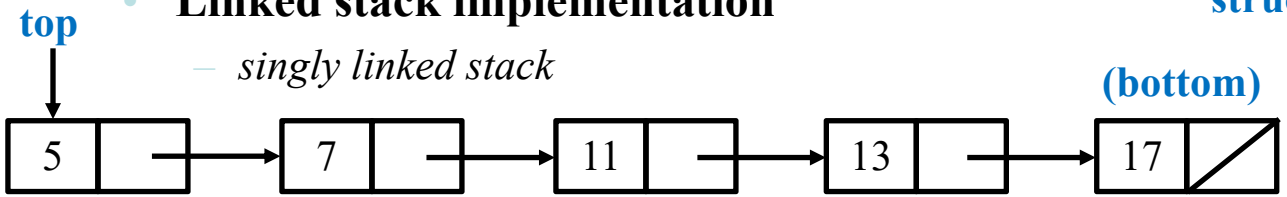
- **Linked stack implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new stack elements as needed
  - *singly linked stack* (already effecient enough)
  - efficient *push & pop*



linked stack after push (or pop) of the node containing 3

Stack

- **Linked stack implementation**
  - *singly linked stack*




```
#ifndef __LSTACK_H__
#define __LSTACK_H__
#include <iostream>
#include <assert.h>
#include "Stack.h"
#include "Link.h" // singly linked stack is already efficient enough

template <typename T> class LStack: public Stack<T>{ // linked stack
private:Link<T>* t; int n; // pointer to top element; stack length
public: LStack(){t=NULL;n=0;}  ~LStack(){clear();}  int length() const{return n;}
        const T& top() const{assert(n!=0 && "Stack is empty");return t->e;}
        void push(const T& item){t=new Link<T>(item,t);n++;}
        T pop(){assert(t!=NULL && "Stack is empty");T it=t->e;
                Link<T>* tmp=t->next;delete t;t=tmp;n--;return it;}
        void clear(){while(t!=NULL){Link<T>* tmp=t;t=t->next;delete tmp;} n=0;}
        void S(){T* en=new T[n];Link<T>* h=t;for(int i=0;i<n;i++){en[i]=h->e;h=h->next;}
                for(int i=n;i;)std::cout<<en[--i]<<' ';std::cout<<"|TOP\n";delete[] en;}
};
#endif
```

Stack

- **Linked stack implementation**
  - *singly linked stack*

```
 g++ demoLStack.cpp -o _a ; ./_a
Empty stack: |TOP
Push 1 into stack: 1 |TOP
Push 2 into stack: 1 2 |TOP
Push 3 into stack: 1 2 3 |TOP
Push 4 into stack: 1 2 3 4 |TOP
Stack currently has 4 elements
Pop 4 from stack: 1 2 3 |TOP
Pop 3 from stack: 1 2 |TOP
Get 2 from stack: 1 2 |TOP
Stack currently has 2 elements
Push 13 into stack: 1 2 13 |TOP
Stack currently has 3 elements
Empty stack: |TOP
Push one into stack: one |TOP
Push two into stack: one two |TOP
Push three into stack: one two three |TOP
Stack currently has 3 elements
Pop three from stack: one two |TOP
Get two from stack: one two |TOP
Stack currently has 2 elements
Push algorithms into stack: one two algorithms |TOP
Stack currently has 3 elements
```

```
#include <iostream>
#include "LStack.h"
using namespace std;
int main(){LStack<int> ai;cout<<"Empty stack: ";ai.S();for(int i=1;i<=4;i++){
                cout<<"Push "<<i<<" into stack: ";ai.push(i);ai.S();}
        cout<<"Stack currently has "<<ai.length()<<" elements\n";
        int e=ai.pop();cout<<"Pop "<<e<<" from stack: ";ai.S();
        e=ai.pop();cout<<"Pop "<<e<<" from stack: ";ai.S();
        e=ai.top();cout<<"Get "<<e<<" from stack: ";ai.S();
        cout<<"Stack currently has "<<ai.length()<<" elements\n";
        cout<<"Push "<<13<<" into stack: ";ai.push(13);ai.S();
        cout<<"Stack currently has "<<ai.length()<<" elements\n";

        LStack<const char*> as;cout<<"Empty stack: ";as.S();
        cout<<"Push one into stack: ";as.push("one");as.S();
        cout<<"Push two into stack: ";as.push("two");as.S();
        cout<<"Push three into stack: ";as.push("three");as.S();
        cout<<"Stack currently has "<<as.length()<<" elements\n";
        const char* ec=as.pop();cout<<"Pop "<<ec<<" from stack: ";as.S();
        ec=as.top();cout<<"Get "<<ec<<" from stack: ";as.S();
        cout<<"Stack currently has "<<as.length()<<" elements\n";
        cout<<"Push algorithms into stack: ";as.push("algorithms");as.S();
        cout<<"Stack currently has "<<as.length()<<" elements\n";return 0;
}
```

Queue

- **Generic list => generic queue (first-in-first-out i.e. FIFO)**
  - a **finite**, **ordered** sequence of data items known as *elements*
    - *ordered* means each element has a *position* in the list, namely a list is a set of elements with *ordinal numbers*
  - *length* (empty if 0)
  - *position* ~~*& current position*~~ => *head* & *tail*
  - *get* (~~current element~~) => *head*
  - *head* & *tail*
  - *next* & *prev*(ious)
  - ~~*move* (to start, end, or any position)~~
  - ~~*insert &*~~ *append* (=> *tail*)
  - *remove* => *head*
  - *clear*

structure abstraction

Queue

- **Generic list => generic queue (first-in-first-out i.e. FIFO)**
  - a **finite**, **ordered** sequence of data items known as *elements*
    - *ordered* means each element has a *position* in the list, namely a list is a set of elements with *ordinal numbers*
  - *length* (empty if 0)
  - *position* & *get* & *head* => ***head***
  - *append* & *next* & *tail* => ***enqueue***
  - *remove* & *next* & *head* => ***dequeue***
  - *clear*

structure abstraction

Queue

- Generic queue (first-in-first-out i.e. FIFO)

structure abstraction

```
#ifndef __QUEUE_H__
#define __QUEUE_H__
template <typename T> class Queue{ // Queue template (ADT)
private:void operator=(const Queue&){} // protect assignment (reflect why)
        Queue(const Queue&){} // protect copy constructor (reflect why)
public: Queue(){} // default constructor
        virtual ~Queue(){} // base destructor
        virtual int length() const=0; // return the number of queue elements
        virtual const T& head() const=0; // copy of the head/front element
        virtual void enqueue(const T&)=0; // add to queue tail
        virtual T dequeue()=0; // remove & return the head/front element
        virtual void clear()=0; // make queue empty
        virtual void S()=0; // show queue elements
};
```

Queue

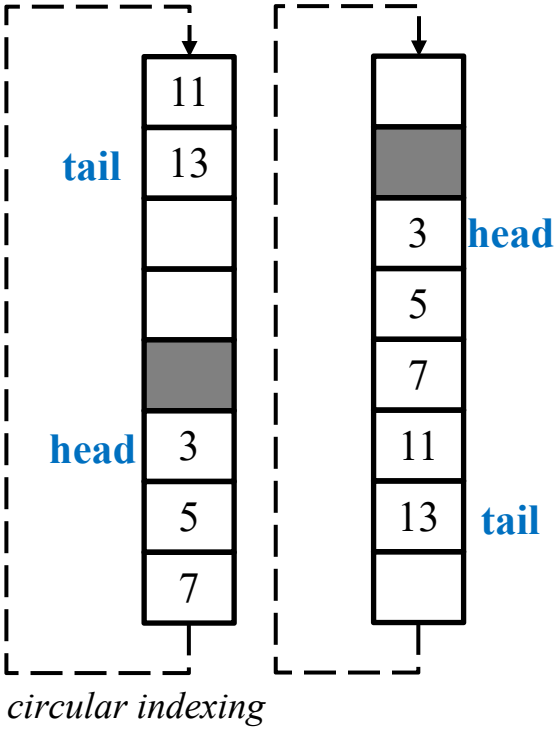
- **Array-based queue implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - *length* (empty if 0)
  - *position* & *get* & *head* => ***head***
  - *append* & *next* & *tail* => ***enqueue***
  - *remove* & *next* & *head* => ***dequeue***
  - *clear*

structure abstraction

Queue

- **Array-based queue implementation**

structure abstraction



circular indexing

```
#include <iostream>
#include <assert.h>
#include "Queue.h"
#define QUEUE_DEFAULT_N 1000
template <typename T> class AQueue: public Queue<T>{ // array-based queue
private:int maxN; // maximum allowable number of queue elements + 1
        int h, t; // index of head (front) element, tail (rear) element
        T* e; // array holding queue elements
public: AQueue(int ni=QUEUE_DEFAULT_N){maxN=ni+1;t=0;h=1;e=new T[maxN];}
        ~AQueue(){delete[] e;} // destructor: deallocate array space
        int length() const{return (t+maxN-h+1)%maxN;} // circular indexing
        const T& head() const{assert(length()!=0 && "Q is empty");
                return e[h];} // copy of the head/front element
        void enqueue(const T& it){assert(((t+2)%maxN)!=h && "Q is full");
                t=(t+1)%maxN;e[t]=it;} // add to queue tail
        T dequeue(){assert(length()!=0 && "Q is empty");T it=e[h];h=(h+1)%maxN;
                return it;} // remove & return the head/front element
        void clear(){t=0;h=1;} // re-initialize
        // t==(h-1 % maxN) is reserved for the empty queue case
        // so t==(h-2 % maxN) means the queue is full
        // this is why maxN = maximum allowable number + 1
        void S(){int t2=t<h-1?t+maxN:t;for(int i=h;i<=t2;i++)
                std::cout<<e[i%maxN]<<' ';std::cout<<'\n';}
};
```

Queue

structure abstraction

- **Array-based queue implementation**

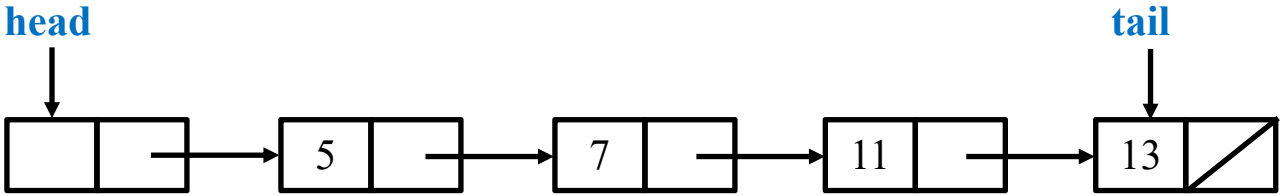
```
 g++ demoAQueue.cpp -o _a; ./_a
Empty queue:
Enqueue 5: 5
Enqueue 6: 5 6
Enqueue 7: 5 6 7
Enqueue 8: 5 6 7 8
Enqueue 9: 5 6 7 8 9
Queue currently has 5 elements
Dequeue 5: 6 7 8 9
Dequeue 6: 7 8 9
Get 7 from queue: 7 8 9
Queue currently has 3 elements
Enqueue 11: 7 8 9 11
Enqueue 13: 7 8 9 11 13
Queue currently has 5 elements
Empty stack:
Enqueue one: one
Enqueue two: one two
Queue currently has 2 elements
Dequeue one: two
Get two: two
Queue currently has 1 elements
Enqueue algorithms: two algorithms
Queue currently has 2 elements
```

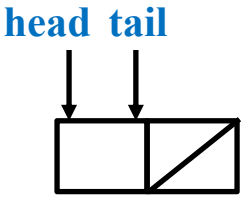
```
#include <iostream>
#include "AQueue.h"
using namespace std;
int main(){AQueue<int> ai(5);cout<<"Empty queue: ";ai.S();for(int i=5;i<=9;i++){
                cout<<"Enqueue "<<i<<": ";ai.enqueue(i);ai.S();}
        cout<<"Queue currently has "<<ai.length()<<" elements\n";
        int e=ai.dequeue();cout<<"Dequeue "<<e<<": ";ai.S();
        e=ai.dequeue();cout<<"Dequeue "<<e<<": ";ai.S();
        e=ai.head();cout<<"Get "<<e<<" from queue: ";ai.S();
        cout<<"Queue currently has "<<ai.length()<<" elements\n";
        cout<<"Enqueue "<<11<<": ";ai.enqueue(11);ai.S();
        cout<<"Enqueue "<<13<<": ";ai.enqueue(13);ai.S();
        cout<<"Queue currently has "<<ai.length()<<" elements\n";

        AQueue<const char*> as;cout<<"Empty stack: ";as.S();
        cout<<"Enqueue one: ";as.enqueue("one");as.S();
        cout<<"Enqueue two: ";as.enqueue("two");as.S();
        cout<<"Queue currently has "<<as.length()<<" elements\n";
        const char* ec=as.dequeue();cout<<"Dequeue "<<ec<<": ";as.S();
        ec=as.head();cout<<"Get "<<ec<<": ";as.S();
        cout<<"Queue currently has "<<as.length()<<" elements\n";
        cout<<"Enqueue algorithms: ";as.enqueue("algorithms");as.S();
        cout<<"Queue currently has "<<as.length()<<" elements\n";return 0;
}
```

Queue

structure abstraction

- **Linked queue implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new queue elements as needed
  - *singly linked queue* (already effecient enough)



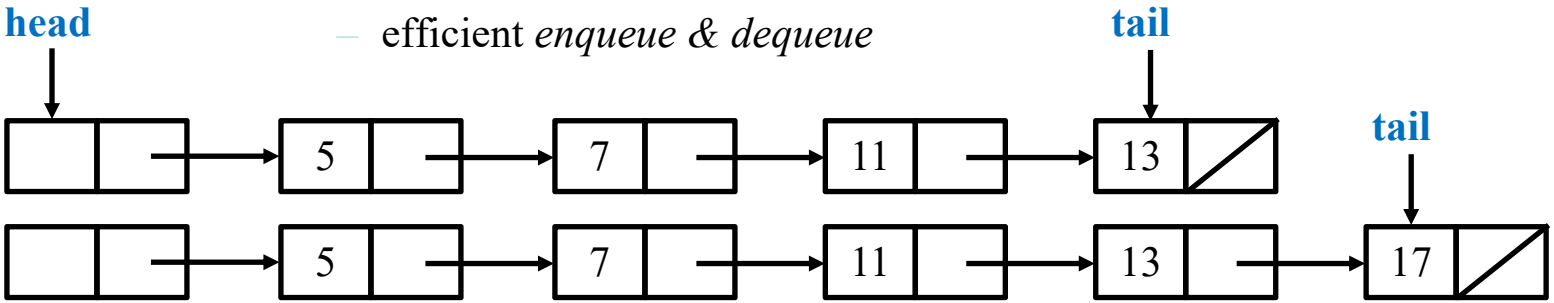


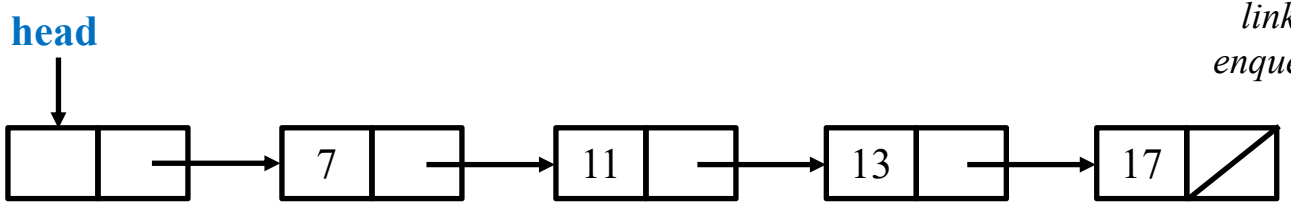
initial state of a linked queue

Queue

structure abstraction

- **Linked queue implementation**
  - a **finite**, **ordered** sequence of data items known as *elements*
  - **dynamic memory allocation**: for new queue elements as needed
  - *singly linked queue* (already effecient enough)
  - efficient *enqueue & dequeue*





linked queue after *enqueue* (or *dequeue*)

Queue

structure abstraction

- **Linked queue implementation**
  - *singly linked queue*

```
#include <iostream>
#include <assert.h>
#include "Queue.h"
#include "Link.h" // singly linked queue is already efficient enough
template <typename T> class LQueue: public Queue<T>{ // linked queue
private:Link<T> *h,*t; // pointers to head (front), tail (rear) link nodes
        int n; // number of elements in queue i.e. queue length
public: LQueue(){h=t=new Link<T>();n=0;}  ~LQueue(){clear();delete h;}
        int length() const{return n;}
        const T& head() const{assert(n!=0 && "Q is empty");
                return h->next->e;} // copy of the head/front element
        void enqueue(const T& it){t->next=new Link<T>(it,NULL);
                t=t->next;n++;} // add to queue tail
// The init condition 'h=t' ensures when Q is empty & 1st node is enqueued,
// 't->next=new Link<T>' can implicitly let h->next=1st node's address,i.e.
// implicitly let head successfully link to tail after enqueue of 1st node
        T dequeue(){assert(n!=0 && "Q is empty");T it=h->next->e;
                Link<T>* tmp=h->next;h->next=tmp->next; if (t==tmp) t=h;
// if dequeue last, Q becomes empty, so don't forget init condition 'h=t'
                delete tmp;n--;return it;} // remove & return head/front
        void clear(){while(h->next!=NULL){t=h;h=h->next;delete t;} t=h;n=0;}
        void S(){Link<T>* c=h->next;while(c!=NULL){
                std::cout<<c->e<<' ';c=c->next;} std::cout<<'\n';}
};
```

Queue

- **Linked queue implementation**
  - *singly linked queue*

```
 g++ demoLQueue.cpp -o _a; ./_a
Empty queue:
Enqueue 5: 5
Enqueue 6: 5 6
Enqueue 7: 5 6 7
Enqueue 8: 5 6 7 8
Enqueue 9: 5 6 7 8 9
Queue currently has 5 elements
Dequeue 5: 6 7 8 9
Dequeue 6: 7 8 9
Get 7 from queue: 7 8 9
Queue currently has 3 elements
Enqueue 11: 7 8 9 11
Enqueue 13: 7 8 9 11 13
Queue currently has 5 elements
Empty stack:
Enqueue one: one
Enqueue two: one two
Queue currently has 2 elements
Dequeue one: two
Get two: two
Queue currently has 1 elements
Enqueue algorithms: two algorithms
Queue currently has 2 elements
```

```
#include <iostream>
#include "LQueue.h"
using namespace std;
int main(){LQueue<int> ai;cout<<"Empty queue: ";ai.S();for(int i=5;i<=9;i++){
                cout<<"Enqueue "<<i<<": ";ai.enqueue(i);ai.S();}
        cout<<"Queue currently has "<<ai.length()<<" elements\n";
        int e=ai.dequeue();cout<<"Dequeue "<<e<<": ";ai.S();
        e=ai.dequeue();cout<<"Dequeue "<<e<<": ";ai.S();
        e=ai.head();cout<<"Get "<<e<<" from queue: ";ai.S();
        cout<<"Queue currently has "<<ai.length()<<" elements\n";
        cout<<"Enqueue "<<11<<": ";ai.enqueue(11);ai.S();
        cout<<"Enqueue "<<13<<": ";ai.enqueue(13);ai.S();
        cout<<"Queue currently has "<<ai.length()<<" elements\n";

        LQueue<const char*> as;cout<<"Empty stack: ";as.S();
        cout<<"Enqueue one: ";as.enqueue("one");as.S();
        cout<<"Enqueue two: ";as.enqueue("two");as.S();
        cout<<"Queue currently has "<<as.length()<<" elements\n";
        const char* ec=as.dequeue();cout<<"Dequeue "<<ec<<": ";as.S();
        ec=as.head();cout<<"Get "<<ec<<": ";as.S();
        cout<<"Queue currently has "<<as.length()<<" elements\n";
        cout<<"Enqueue algorithms: ";as.enqueue("algorithms");as.S();
        cout<<"Queue currently has "<<as.length()<<" elements\n";return 0;
}
```

THANK YOU

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY