Contents

Part I

C++ Generals

Most of C++ generals content are from my *Starting out with C++* notes.

Chapter 1

Inheritance, Polymorphism and Virtual Functions

1.1 Inheritance

1.1.1 Concept

- Inheritance allows a new class to be based on an existing class;
- The new class inherits all the member variables and functions of the class it based on
- The new class won't inherit constructors and destructor of the class it based on

1.1.2 Origin of Inheritance: Generalization and Specialization

In the real world you can find many objects that are specialized versions of other more general objects. For example, dog is a specialized object of animal.

Inheritance allows abstraction of this kind of relation: creating new class that is based on an existing class. When one object is a specialized version of another object, there is an "is a" relationship between them, for example: a tree is a plant, a dog is an animal.

When an "is a" relationship exists between classes, it means that the specialized class has all of the characteristics of the general class, plus additional characteristics that make it special. In object-oriented programming, inheritance is used to create an "is a" relationship between classes.

Inheritance involves a base class and a derived class. The base class is the general class and the derived class is the specialized class. The derived class has following features:

- it inherits the member variables and member functions of the base class without any of them being rewritten (including private and public members)
- it does not inherit constructors and destructor

- it can be added with new member variables and functions, making it more specialized than the base class

1.1.3 Syntax

Suppose you have a `Base` class. Now you want to make a derived class from `Base` named `Derived`. You declare the `Derived` class in following way:

```
class Derived : public Base {
  // class definition
  // goes here
};
```

This declaration tells you that, `Derived` is a `Base`. Similarly, we can have:

```
class Dog : public Animal; // Dog is an Animal
class Tree : public Plant; // Tree is a plant
class Ginkgo : public Tree; // Ginko is a Tree
```

You have to `#include` the base class header file in the derived class's header file.

The word, public which precedes the name of the Base class, is the base class access specification. It affects how the members of the Base class are inherited by the Derived class. When you create an object of a Derived class, you can think of it as being built on top of an object of the Base class. The members of the Base class object becomes members of the Derived class object automatically. How the Base class members appear in the Derived class is determined by the base class access specification. Base class access specification will be covered in detail in next section (Class Access).

If you declare the base class access specification as public, you can access the public members of the Base class without any additional declarations. For example, you can call public member functions of the Base class. Although you can't access the private member of the Base class directly in the Derived class, you can access them via the interface defined in public of the Base class.

Inheritance does not work in reverse. It is not possible for a base class to call a member function of a derived class.

1.1.4 Class Access Specifications

There are three class access specifications between a base class and a derived class:

```
class Derived : private Base;
class Derived : protected Base;
class Derived : public Base;
```

Details omitted

1.1.5 Constructors and Destructors in Base and Derived Classes

1.1.5.1 Calling order

Suppose you have a Derived class and a Base class. Now you are defining an object of a Derived class. Then you exit the program so the Derived object is destroyed. The order of calling constructors and destructor is as follows:

- constructor of base class
- constructor of derived class
- destructor of derived class
- destructor of base class

1.1.5.2 Passing arguments to base class constructors

Consider following case:

```
class Derived : public Base
```

When you are creating a Derived object, you will first call the constructor of Base class. In order to pass arguments to Base class constructor, you have to let the Derived class constructor pass arguments to the Base class constructor.

Now, suppose the Base class has a constructor that needs two arguments: length and width. And Derived class has a constructor that needs one argument: height. When you write the parameter list for constructor of Derived class, you can also include the parameter for constructor of Base class. Here is how you should write when you **implement the constructor**.

```
Derived::Derived() : Base() {
  // implementation of default constructor
}

Derived::Derived(double len, double w, double h) : Base(len, w) {
  // implementation of Derived(len, w, h)
}
```

Pay attention that, calling and passing parameter to `Base()` is when you **IMPLEMENT** `Derived` class's constructor. When declaring constructor of `Derived` class, you write in the normal way:

```
Derived();
Derived(double len, double w, double h);
```

however, if you are writting the constructor inline, you have to write it in .h file.

Note 1 The order of the parameter in Derived class constructor is not required. You just need to put the argument in the followed Base class constructor in right order.

Note 2 The Base class constructor is always executed before the Derived class constructor. It will take the argument it needs in the argument list of Derived class constructor, and executes. When the Base constructor finishes, the Derived class constructor is then executed.

Note 3 If the Base class has no default constructor, then the Derived class must have a constructor that calls one of the Base class constructors. Because normally, if all the Derived class constructor don't call Base class constructor, when a Derived class object is made, the default Base class constructor will be called.

1.1.5.3 Redefining `Base` class functions

You can redefine a Base class member function in its Derived class. Its like an overloaded member function (because the function has the same name), however, there is a distinction between redefining a function and overloading a function:

- Overloading functions must have **DIFFERENT** function signature (same function name, but different parameter list)
- Redefining function must have **SAME** function signature. Redefining happens when a Derived class has a function with the same name and same parameter list as a Base class function.

Suppose you have following classes defined:

```
class Derived : public Base
```

There is a function `func(int num)` in Base class. Now you want to redefine `func()` in Derived class. You can directly declare it in header file of Derived class:

```
class Derived : public Base {
private:
  // declaration of private member
public:
  void func(int num2) { // define the func() directly

    int num = num2 / 2;
    Base::func(num); // you can call the original func() in Base using
    ↪  scope resolution operator

    // additional definition
    // goes here
  }
};
```

If you redefine the function. You have to use the scope resolution operator :: to call the original function in Base class. C++ cast static binding to re-defined member function. That is, function call and the corresponding function is bound together when compiling.

Part II

Data Structures

Chapter 2

Vector

Chapter 3

List

Some portion of this section is copied from my *Starting out with C++* notes. The implementation is from my project 2 of COP 4530 course.

3.1 General Idea

Suppose you design a program that will dynamically allocate data structures, and you want to manage them. You can use linked list to do this job.

Dynamically allocated data structures may be linked together in memory to form a chain (linked list), each dynamically allocated data structure in that linked list is called **node**. A linked list can easily grow or shrink in size. Also, compared with vector, it is very efficient for a linked list to insert an element into the middle of the list, or delete an element that is in the middle. For a vector, to access a middle element you have to move all the element after that position, while for a linked list, none of the other nodes have to be moved.

A linked list is a series of connected nodes, where each node is a data structure. By saying "connected", it means that there is a pointer in each node that points to next node, thus the term "connected". Data members hold the data that this node contain (one or more members). In addition to the data, each node contains a pointer, which can point to another node (the next node). A linked list is formed when each node points to the next node. A **doubly linked list** is when each node contains two pointers, which can link to the previous node and the next node.

3.2 Simple Implementation

In this section, we'll implemente a doubly linked list, which has similar interface as the STL list. We'll also implement a nested iterator class in the list.

3.2.1 Outline of List Class

We'll encapsulate three types in `List` class: 1. the node structure; 2. the nested constant iterator; 3. the nested iterator. Our `List` class will also support similar interface with STL list. Aside from member functions, our `List` class has the following data members:

```
int theSize; // number of elements in the list
Node* head; // store head node
Node* tail; // store the tail node
```

To scope a `List` structure, we'll use two `Node*` type variable: `head` and `tail`. They won't store any data. Their purpose is to serve as delimiter of our `List` object.

We'll also define a namespace named `cop4530`, and we will put the `List` class into namespace `cop4530`.

The complete header is listed below:

```
#ifndef DL_LIST_H
#define DL_LIST_H
#include <iostream>

namespace cop4530 {

  template <typename T>
    class List {
      private:
        // nested Node class
        struct Node {
        // declared later
        };

      public:
        //nested const_iterator class
        class const_iterator {
        // declared later
        };

        // nested iterator class
        class iterator : public const_iterator {
        // declared later
        };

      public:
        // constructor, desctructor, copy constructor
        List(); // default zero parameter constructor
        List(const List &rhs); // copy constructor
```

```
30          List(List && rhs); // move constructor
31          // num elements with value of val
32          explicit List(int num, const T& val = T{});
33          // constructs with elements [start, end)
34          List(const_iterator start, const_iterator end);
35
36          ~List(); // destructor
37
38          // copy assignment operator
39          const List& operator=(const List &rhs);
40          // move assignment operator
41          List & operator=(List && rhs);
42
43          // member functions
44          int size() const; // number of elements
45          bool empty() const; // check if list is empty
46          void clear(); // delete all elements
47          void reverse(); // reverse the order of the elements
48
49          T &front(); // reference to the first element
50          const T& front() const;
51          T &back(); // reference to the last element
52          const T & back() const;
53
54          void push_front(const T & val); // insert to the beginning
55          void push_front(T && val); // move version of insert
56          void push_back(const T & val); // insert to the end
57          void push_back(T && val); // move version of insert
58          void pop_front(); // delete first element
59          void pop_back(); // delete last element
60
61          void remove(const T &val); // remove all elements with value = val
62
63          // print out all elements. ofc is deliminitor
64          void print(std::ostream& os, char ofc = ' ') const;
65
66          iterator begin(); // iterator to first element
67          const_iterator begin() const;
68          iterator end(); // end marker iterator
69          const_iterator end() const;
70
71          iterator insert(iterator itr, const T& val); // insert val ahead of
            ↪ itr
72          iterator insert(iterator itr, T && val); // move version of insert
73
```

```
74          iterator erase(iterator itr); // erase one element
75          iterator erase(iterator start, iterator end); // erase [start, end)
76
77
78        private:
79          int theSize; // number of elements
80          Node *head; // head node
81          Node *tail; // tail node
82
83          void init(); // initialization
84      };
85
86    // overloading comparison operators
87    template <typename T>
88    bool operator==(const List<T> & lhs, const List<T> &rhs);
89
90    template <typename T>
91    bool operator!=(const List<T> & lhs, const List<T> &rhs);
92
93    // overloading output operator
94    template <typename T>
95    std::ostream & operator<<(std::ostream &os, const List<T> &l);
96
97    // include the implementation file here
98    #include "List.hpp"
99
100 } // end of namespace 4530
101
102 #endif
```

3.2.2 Node Structure

The structure is defined as the basic unit of our list. List is composed of multiple nodes that are connected with each other. Thus, one node structure should have three members: one for storing data, another will store the address of previous node and next node. The code for the Node structure is as follows:

```
1  struct Node {
2    T data;
3    Node* prev;
4    Node* next;
5
6    // copy constructor
7    Node(const T& d = T{}, Node* p = nullptr, Node* n = nullptr) : data{d},
     ↪  prev{p}, next{n} {}
```

```
8
9     // move constructor
10    Node(T&& d = T{}, Node* p = nullptr, Node* n = nullptr) : data{d},
      ↪  prev{p}, next{n} {}
11 };
```

In the constructor, we give default argument. Notice this line:

```
const T& d = T{}
```

this is actually calling the zero parameter constructor of class `T` to build an object `d`, this default object will be used as the default data member stored by the node.

3.2.3 Iterator Support

Some containers can just use pointer to do jobs an iterator do. For example, `std::vector`. This is because, internally, elements of vector is stored in an array, their internal memory address is consecutive. Thus, you can use operators such `++`, `--` to navigate through the container. And other operations such as dereference (`*`), comparison (`==`), etc.

For containers whose elements are not stored consecutively in memory, we can't use the above mentioned feature. Since they are very handy to use (to navigate the whole container), and more importantly, since we want to provide a universal operating interface for all the containers, we want to implement such data type to these containers too. An iterator class is constructed just to do that.

Back to our specific problem, we want to design an iterator class for our `List` class. We will implement its template inside the `List` class as a public member (so we can use it to create the actual iterator object from outside of the `List` class).

We'll define two versions of iterator: a constant one and a non-constant one. The constant one can't be used to modify the item it referenced to. These two iterator share most of the member functions, so we may want to declare one as **base** class, the other one as **derived** class. We'll provide the header declaration here. The implementation of member function of iterator will be provided later.

3.2.3.1 Nested `const_iterator`

We'll first define a constant iterator as the base class. Later, we'll declare the non-constant version of iterator as derived class, based on our constant iterator.

The **CORE** or the iterator class is the protected member: a pointer whose type is the type you want the iterator to reference. After all, the iterator should act like a pointer that can dereference each element in the container, while navigate through the container. So we have the following data member in `protected` section of our `const_iterator` class:

```
Node* current; // pointer to node in list
```

We declare a function interface to handle retriving reference operation:

```
T& retrieve() const;
```

The specifier `const` simpliy suggests that this function will not change anything inside `const_iterator` class (pay attention that it does not mean this function cannot change the `Node` element it reference to).

We also have a protected constructor which accepts a pointer to `Node` to create a `const_iterator` type:

```
const_iterator(Node* p);
```

At last, we declare class `List<T>` as the friend our this `const_iterator` class, which means class `List<T>` can access the private and protected member of `const_iterator` class. Some routines in `List` may need to access:

```
friend class List<T>;
```

To provide `const_iterator` class the ability to dereference an element, we provide the routine for `operator*()`:

```
const T& operator*() const;
```

Notice that, the return type is a constant left value reference type. This is because we are implementing the `operator~()` of constant iterator, so the `Node` being dereferenced should not be changed.

The next property we wish to add to our iterator class is the ability to navigate through the container. For pointers to array elements, we can use `operator++()`, `operator--()` to navigate around the array (because memories of the elements are stored adjacently). For our iterator class, we have to manually implement this behavior. Also, we want to enable comparison operators so we can use iterators to check whether the boundary of our container is reached (`itr == container.end()`). These routines are declared as follows:

```
// increment/decrement operators
const_iterator & operator++();
const_iterator operator++(int);
const_iterator & operator--();
const_iterator operator--(int);

// comparison operators
bool operator==(const const_iterator &rhs) const;
bool operator!=(const const_iterator &rhs) const;
```

The whole declaration of `const_iterator` is as follows:

```
1  class const_iterator {
2    public:
3      const_iterator(); // default zero parameter constructor
4      const T & operator*() const; // operator*() to return element
5
6      // increment/decrement operators
```

```
7       const_iterator & operator++();
8       const_iterator operator++(int);
9       const_iterator & operator--();
10      const_iterator operator--(int);
11
12      // comparison operators
13      bool operator==(const const_iterator &rhs) const;
14      bool operator!=(const const_iterator &rhs) const;
15
16    protected:
17      Node *current; // pointer to node in List
18      T & retrieve() const; // retrieve the element refers to
19      const_iterator(Node *p); // protected constructor
20
21      friend class List<T>;
22  };
```

3.2.3.2 Nested `iterator`

We'll declare our `iterator` class as derived class of `const_iterator` since they share a lot of common functions. To declare a derived class from `const_iterator`, we use:

```
class iterator : public const_iterator {

};
```

Since this is not constant iterator, we must **redefine** the function of `operator*()` to provide a version that can return reference (`T&`), rather than the constant reference (`const T&`) brought by the original `operator*()` function defined in base class `const_iterator`. We also wish to provide the constant version of the function, so we have:

```
T& operator*(); // return reference
const T& operator*(); // return constant reference
```

Also, all functions that should have different return types should be redefined:

```
iterator & operator++();
iterator operator++(int);
iterator & operator--();
iterator operator--(int);
```

At last, we declare the constructor of our `iterator` class:

```
iterator(Node* p);
```

3.2.4 `const_iterator` Implementation

3.2.4.1 Zero parameter `const_iterator()`

Initialize `current` with `nullptr`.

```
template <typename T>
List<T>::const_iterator::const_iterator() : current {nullptr} {}
```

Notice that the use of scope resolution operator. We are defining a function in `const_iterator` class, which is in `List` class.

3.2.4.2 One parameter `const_iterator`

It accepts a pointer to a `Node`, will initialize `current` with this pointer.

```
template <typename T>
List<T>::const_iterator::const_iterator(Node *p) : current{ p } {};
```

3.2.4.3 `operator*()`

Function `retrieve()` will be called to get the reference.

```
//returns a reference to the corresponding element in the list by calling
↪  retrieve() member function
template <typename T>
const T& List<T>::const_iterator::operator*() const {
  return retrieve();
}
```

3.2.4.4 `operator++()`

No `int` in the parenthese, so this is an prefix increment operator. We want to move the iterator to the **NEXT** node. We know that `Node.next` is storing the memory address of next node, that's how we move to it:

```
//increment_prefix
template <typename T>
typename List<T>::const_iterator& List<T>::const_iterator::operator++() {
  current = current->next;
  return *this;
}
```

Also notice that a keyword `typename` is in front of the `List<T>::const_iterator`, this tells compiler that, `List<T>::const_iterator` is a name of a type, which is the return type of the function defined in this line later.

3.2.4.5 operator++(int)

Postfix increment. We need to return the iterator before the increment. Since the function will return during this process, we can't first return and then do the increment. We can only make a copy of it and return later. This is why postfix increment may have less efficiency compared with prefix increment: another copy is necessary.

```
//increment_postfix
template <typename T>
typename List<T>::const_iterator List<T>::const_iterator::operator++(int) {
  const_iterator old = *this;
  ++(*this);
  return old;
}
```

3.2.4.6 operator--()

Similar with operator++():

```
template <typename T>
typename List<T>::const_iterator& List<T>::const_iterator::operator--() {
  current = current->prev;
  return *this;
}
```

3.2.4.7 operator--(int)

Similar with operator++():

```
template <typename T>
typename List<T>::const_iterator List<T>::const_iterator::operator--(int) {
  const_iterator old = *this;
  --(*this);
  return old;
}
```

3.2.4.8 operator==()

If two pointers in two iterators are the same, we say two iterators are equal:

```
template <typename T>
bool List<T>::const_iterator::operator==(const const_iterator& rhs) const {
  return current == rhs.current;
}
```

3.2.4.9 operator!=()

```
template <typename T>
bool List<T>::const_iterator::operator!=(const const_iterator& rhs) const {
```

```
3   return !(*this == rhs);
4 }
```

3.2.4.10 `retrieve()`

The internal `retrieve()` function will return a non-constant reference to the data the node holds. So this retrieve function can be re-used by the non-constant iterator. Function `const_iterator::operator*()` will return a constant reference, so don't worry.

```
1 template <typename T>
2 T& List<T>::const_iterator::retrieve() const {
3   return current->data;
4 }
```

3.2.5 `iterator` Implementation

This section is just about redefining functions so reference, rather than constant reference, will be returned by member functions of `iterator` class.

3.2.5.1 Zero parameter `iterator()`

Do nothing, constructor of base class `const_iterator()` will initialize `current` with `nullptr`.

```
1 template <typename T>
2 List<T>::iterator::iterator() {}
```

3.2.5.2 One parameter `iterator()`

Will call base class's one parameter constructor and pass the parameter to it.

```
1 template <typename T>
2 List<T>::iterator::iterator(Node *p) : const_iterator{ p } {};
```

3.2.5.3 `operator*()`

Will call `const_iterator::retrieve()` to return a reference:

```
1 T& List<T>::iterator::operator*() {
2   return const_iterator::retrieve();
3 }
```

3.2.5.4 constant `operator*()`

Since `const_iterator::operator*()` has been redefined, when we want constant reference be returned, we'll not call `const_iterator::operator*()` automatically, so we have to also redefine a version of `iterator::operator*()` to return constant reference. This is useful when other routines requires a constant reference from a non-constant iterator.

```
template <typename T>
const T& List<T>::iterator::operator*() const {
  return const_iterator::operator*();
}
```

3.2.5.5 operator++()

```
//prefix
template <typename T>
typename List<T>::iterator& List<T>::iterator::operator++() {
  this->current = this->current->next;
  return *this;
}
```

3.2.5.6 operator++(int)

```
// postfix
template <typename T>
typename List<T>::iterator List<T>::iterator::operator++(int) {
  iterator old = *this;
  ++(*this);
  return old;
}
```

3.2.5.7 operator--()

```
// prefix
template <typename T>
typename List<T>::iterator& List<T>::iterator::operator--() {
  this->current = this->current->prev;
  return *this;
}
```

3.2.5.8 operator--(int)

```
// postfix
template <typename T>
typename List<T>::iterator List<T>::iterator::operator--(int) {
  iterator old = *this;
  --(*this);
  return old;
}
```

3.2.6 List() [zero]

Will call `init()` to initialize list member variables

```
template <typename T>
List<T>::List() {
  init();
}
```

3.2.7 List() [copy]

Working steps:

- Call `init()` to initialize list
- use a range based for loop to traverse `rhs`, and use `push_back()` routine to add to this list.

Code:

```
template <typename T>
List<T>::List(const List& rhs) {
  init();
  for (auto& x : rhs)
    push_back(x);
}
```

3.2.8 List() [move]

```
//move constructor
template <typename T>
List<T>::List(List&& rhs) : theSize{ rhs.theSize }, head{ rhs.head }, tail{
↪  rhs.tail } {
  rhs.theSize = 0;
  rhs.head = nullptr;
  rhs.tail = nullptr;
}
```

3.2.9 List() [two parameter]

```
//constructor which accepts number of elements (num) and value for each
↪  element
//Construct a list with num elements, all initialized with value val.
template <typename T>
List<T>::List(int num, const T& val) {
  init();
  for (int i = 0; i < num; ++i)
    push_back(val);
}
```

3.2.10 List() [accept two iterators]

It accepts two iterators indicating the range of a list, and will construct a copy of this range of list.

```
//constructs with elements [start, end)
template <typename T>
List<T>::List(const_iterator start, const_iterator end) {
  init();
  for (const_iterator itr = start; itr != end; ++itr)
    push_back(*itr);
}
```

3.2.11 ~List()

Call `clear()` routine.

```
//destructor
template <typename T>
List<T>::~List() {
  clear();
  delete head;
  delete tail;
}
```

3.2.12 operator=() [copy]

Will apply copy constructor and `std::swap()`:

```
template <typename T>
const List<T>& List<T>::operator=(const List& rhs) {
  List copy = rhs; //apply copy constructor
  std::swap(*this, copy);
  return *this;
}
```

3.2.13 operator=() [move]

Apply `std::swap()`:

```
template <typename T>
List<T>& List<T>::operator=(List&& rhs) {
  std::swap(theSize, rhs.theSize);
  std::swap(head, rhs.head);
  std::swap(tail, rhs.tail);

```

```
  return *this;
}
```

3.2.14 size()

```
//return the number of elements in the List
template <typename T>
int List<T>::size() const {
  return theSize;
}
```

3.2.15 empty()

```
//return true if no element is in the list; otherwise, return false
template <typename T>
bool List<T>::empty() const {
  return size() == 0;
}
```

3.2.16 clear()

Use `pop_front()` routine to delete elements one by one.

```
//delete all the elements in the list
template <typename T>
void List<T>::clear() {
  while (!empty())
    pop_front();
}
```

3.2.17 reverse()

```
//reverse the order of the elements in the list
template <typename T>
void List<T>::reverse() {
  iterator itr_insert = end();
  for (int i = 0; i < size() - 1; ++i) {
    itr_insert = insert(itr_insert, std::move(front()));
    pop_front();
  }
}
```

3.2.18 front()

```
//return reference to the first element in the list
//reference
```

```
3  template <typename T>
4  T& List<T>::front() {
5    return *begin();
6  }
```

3.2.19 front() [const]

```
1  template <typename T>
2  const T& List<T>::front() const {
3    return *begin();
4  }
```

3.2.20 back()

```
1  //return reference to the last element in the list
2  //reference
3  template <typename T>
4  T& List<T>::back() {
5    return *(--end());
6  }
```

3.2.21 back() [const]

```
1  template <typename T>
2  const T& List<T>::back() const {
3    return *(--end());
4  }
```

3.2.22 push_front()

Will call `insert()` routine

```
1  //insert the new object as the first element into the list
2  //copy version
3  template <typename T>
4  void List<T>::push_front(const T& val) {
5    insert(begin(), val);
6  }
```

3.2.23 push_front() [move]

```
1  //move version
2  template <typename T>
3  void List<T>::push_front(T&& val) {
4    insert(begin(), std::move(val));
5  }
```

3.2.24 push_back()

Will call `insert()` routine.

```
//push_back(): insert the new object as the last element into the list
//copy version
template <typename T>
void List<T>::push_back(const T& val) {
  insert(end(), val);
}
```

3.2.25 push_back() [move]

```
template <typename T>
void List<T>::push_back(T&& val) {
  insert(end(), std::move(val));
}
```

3.2.26 pop_front()

Will call `erase()` routine

```
//delete the first element
template <typename T>
void List<T>::pop_front() {
  erase(begin());
}
```

3.2.27 pop_back()

Will call `erase()` routine

```
template <typename T>
void List<T>::pop_back() {
  erase(--end());
}
```

3.2.28 remove()

```
//delete all nodes with value equal to val from the list
template <typename T>
void List<T>::remove(const T& val) {
  for (iterator itr = begin(); itr != end(); ++itr) {
    if (*itr == val) {
      itr = erase(itr);
      --itr; // notice the current position of itr
    }
```

```
  }
}
```

3.2.29 print()

```
//print all elements in the list
template <typename T>
void List<T>::print(std::ostream& os, char ofc) const {
  for (const_iterator itr = begin(); itr != end(); ++itr) {
    os << *itr << ofc;
  }
}
```

3.2.30 begin()

```
//return iterator to the first element in the list
//not const version
template <typename T>
typename List<T>::iterator List<T>::begin() {
  return { head->next };
}
```

3.2.31 begin() [const]

```
//const version
template <typename T>
typename List<T>::const_iterator List<T>::begin() const {
  return { head->next };
}
```

3.2.32 end()

```
//return iterator to the tail node (after the last element) in the list
//not const version
template <typename T>
typename List<T>::iterator List<T>::end() {
  return { tail };
}
```

3.2.33 end() [const]

```
//const version
template <typename T>
typename List<T>::const_iterator List<T>::end() const {
```

```
4     return { tail };
5   }
```

3.2.34 insert()

This function accepts two parameters: a iterator `itr` indicates where to insert the value, and a `T` type `val`, which is value to be inserted into the list. A `Node` will be constructed using `val`, and will be inserted into the previous spot where `itr` points to. `theSize` will be updated in the process.

The three steps to insert a node is shown below.

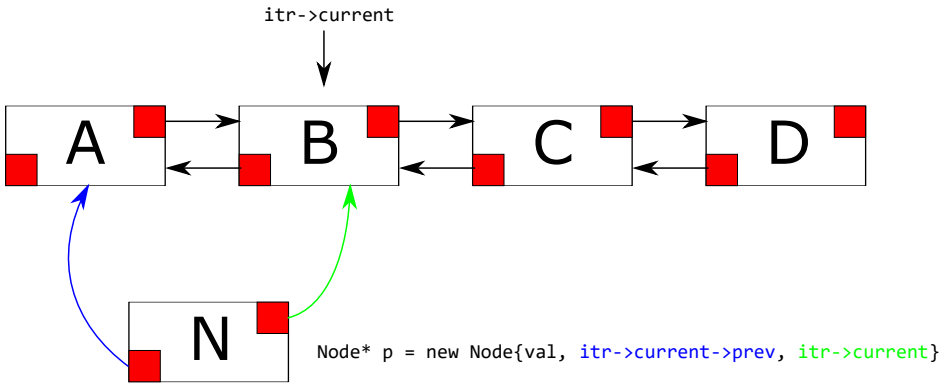

Figure 3.1: Insert a node in a list. Step 1: construct a node

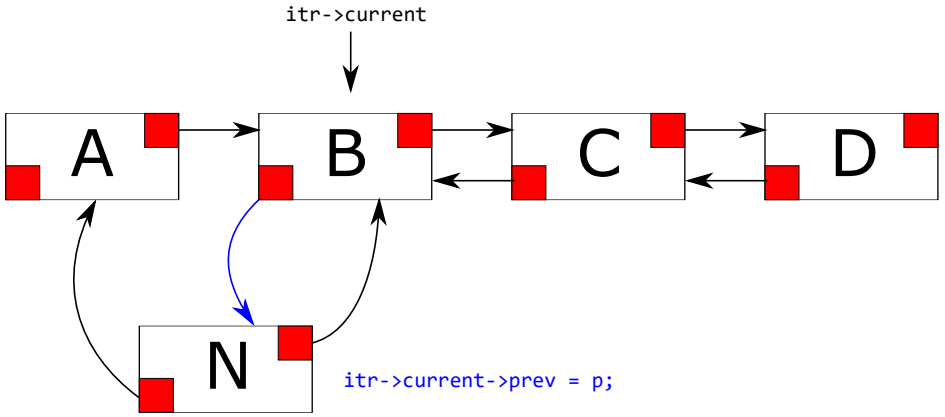

Figure 3.2: Insert a node in a list. Step 2: link node

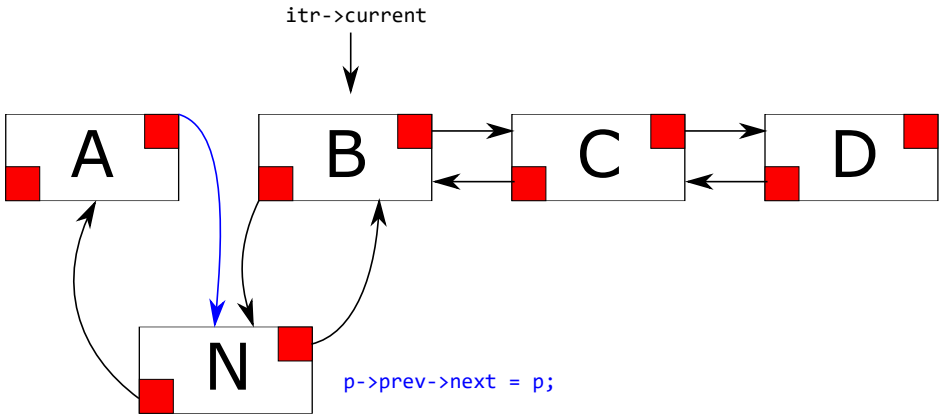

Figure 3.3: Insert a node in a list. Step 3: link node

The code is as follows:

```
template <typename T>
typename List<T>::iterator List<T>::insert(iterator itr, const T& val) {
  // update the size of the list
  theSize++;

  // construct node
  Node* p = new Node{val, itr->current->prev, itr->current};

  // link the node back inti the list
  itr->current->prev = p;
  p->prev->next = p;

  // return an iterator pointing to the inserted element
  // notice that the constructor of iterator which accepts a pointer to a
   ↪  Node has been called implicitly to construct an iterator and return.
  return p;
}
```

Notice that we are calling the following constructor of `iterator`:

```
iterator(Node* p);
```

This is why we need to declare `List<T>` as friend of our iterator class (this constructor is in protected region).

Move version of insert is similar, except for the use of `std::move()`. We'll simplify the above code in just one line:

```
// move version of insert()
template <typename T>
typename List<T>::iterator List<T>::insert(iterator itr, T&& val) {
  theSize++;
```

```
5   return { (*(itr.current)).prev = ((*(itr.current)).prev)->next = new
      ↪  Node{std::move(val), (*(itr.current)).prev, itr.current} };
6 }
```

3.2.35 erase()

This function accepts an iterator. It will delete the node referenced by the iterator and return the iterator to the next node. Content referenced by `itr` will be reclaimed by `delete`. The size of the list will also update. The process of deleting a node in list is shown below.

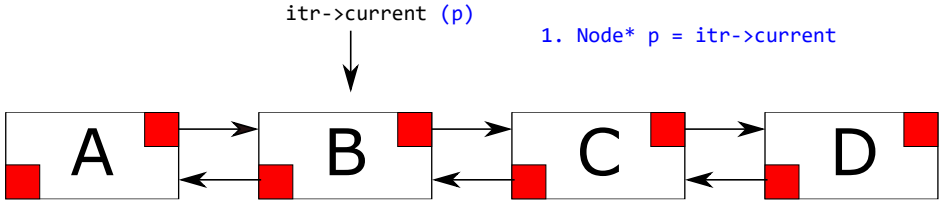

Figure 3.4: Delete a node in a list. Step 1: convenient renaming

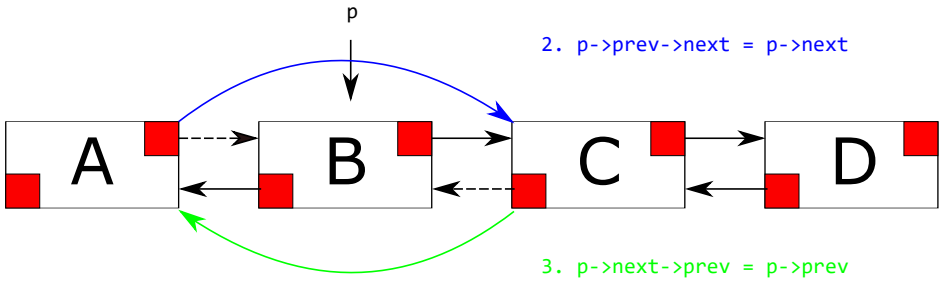

Figure 3.5: Delete a node in a list. Step 2: reconnect node

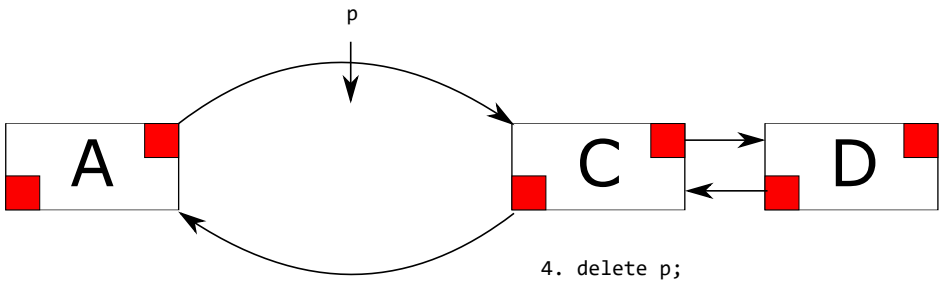

Figure 3.6: Delete a node in a list. Step 3: delete node

Also, pay attention that we want to return an iterator referencing the next node. So we want to keep copy of that first, and return at the end of the function. The code is as follows:

```
1 template <typename T>
2 typename List<T>::iterator List<T>::erase(iterator itr) {
3   Node* p = itr.current;
4   iterator retVal{ p->next };
5   (p->prev)->next = p->next;
6   (p->next)->prev = p->prev;
```

```
7   delete p;
8   theSize--;
9
10  return retVal;
11 }
```

3.2.36 erase() [range based]

We'll call the `erase()` function we have just defined.

```
1  template <typename T>
2  typename List<T>::iterator List<T>::erase(iterator start, iterator end) {
3    for (iterator itr = start; itr != end;)
4      itr = erase(itr);
5
6    return end;
7  }
```

3.2.37 init()

This function will initialize the data member of an empty `List` object. It will connect `head` and `tail` together.

```
1  template <typename T>
2  void List<T>::init() {
3    theSize = 0;
4    head = new Node;
5    tail = new Node;
6    head->next = tail;
7    tail->prev = head;
8  }
```

3.2.38 Nonclass global functions

We also defined some global functions that related to `List` operations. Pay attention that they are not part of the `List` class.

3.2.38.1 operator==()

Two `List` object is said equal, if they have same number of elements and all the corresponding elements are the same.

```
1  template <typename T>
2  bool cop4530::operator==(const List<T>& lhs, const List<T>& rhs) {
3    if (lhs.size() != rhs.size())
4      return false;
5
```

```
6   for (auto itr_lhs = lhs.begin(), itr_rhs = rhs.begin(); itr_lhs !=
    ↪  lhs.end(); ++itr_lhs, ++itr_rhs) {
7     if (*itr_lhs != *itr_rhs)
8       return false;
9   }
10
11  return true;
12
13 }
```

3.2.38.2 operator!=()

Call `operator==()` to finish the work.

```
1 template <typename T>
2 bool cop4530::operator!=(const List<T>& lhs, const List<T>& rhs) {
3   return !(lhs == rhs);
4 }
```

3.2.38.3 operator«()

If implemented globally, `operator«()` may not acces the private member of `List`. I used to declare this function as friend of `List` inside the header of `List` class. But actually, you have defined a public interface `List<T>::print()`, so we can just call that function and pass the `std::ostream` object. Code:

```
1 template <typename T>
2 std::ostream& cop4530::operator<<(std::ostream& os, const List<T>& l) {
3   l.print(os);
4   return os;
5 }
```

Chapter 4

Stack

4.1 General Idea

A stack is a data structure that can manage elements in a **Last-in-First-out** manner, or **LIFO**. Stacks are useful data structures for algorithms that work first with the last saved element of a series. For example, you have a program that looks like this:

```
{
  {
    {
      {
        {
          // the innermost function is doing something here
          // local variable created here will be the last in the stack
          // but will be the first to out the stack
          // because it will be destroyed when coming out from this block
        }
      }
    }
  }
}
```

Stacks can be implemented as an adapter class — it uses another container as underlying storage container, and we implement routines that support the **LIFO** behavior.

4.2 Simple Implementation

In this section, I'll implement a stack template. The default underlying container is `std::deque`, but can be named other. The `Stack` class will be implemented in namespace cop4530.

4.2.1 Outline of `cop4530::Stack`

We declare the `Stack` class as follows:

```
template <typename T, typename Container = std::deque<T>>
class Stack {

};
```

Notice how we declare the default type of container as `std::deque<T>`. The whole outline of `cop4530::Stack` is as follows. When we implement the member functions, we can use the corresponding functions of the underlying container.

```
#ifndef MY_STACK_H
#define MY_STACK_H
#include <deque>
#include <iostream>


namespace cop4530 {

  /***** class template Stack<T, Container> *****/
  template <typename T, typename Container = std::deque<T>>
  class Stack {
    protected:
      //internal container
      Container c;
      //friend of Stack class
      //https://web.mst.edu/~nmjxv3/articles/templates.html
      template <typename A, typename B>
      friend bool operator==(const Stack<A,B>& lhs, const Stack<A,B>& rhs);
      template <typename A, typename B>
      friend bool operator<=(const Stack<A,B>& lhs, const Stack<A,B>& rhs);

    public:
      //constructor, destructor, copy constructor, move constructor
      Stack(); //default zero-argument constructor
      ~Stack(); //destructor
      Stack(const Stack<T, Container>& rhs); //copy constructor
      Stack(Stack<T, Container>&& rhs); //move constructor

      // copy and move assignment operator
      Stack<T, Container>& operator=(const Stack<T, Container>& rhs);
      ↪ //copy assignment operator=
      Stack<T, Container>& operator=(Stack<T, Container>&& rhs); //move
      ↪ assignment operator=
```

```
32
33          // Member functions
34          bool empty() const; //returns true if the Stack contains no elements,
            ↪  and false otherwise
35          void clear(); //delete all elements from the stack
36          void push(const T& x); //adds x to the Stack, copy version
37          void push(T&& x); //adds x to the Stack, move version
38          void pop(); //removes and discards the most recently added element of
            ↪  the Stack
39          T& top(); //returns a reference to the most recently added element of
            ↪  the Stack
40          const T& top() const; //returns a const reference to the most
            ↪  recently added element of the Stack
41          int size() const; //returns the number of elements stored in the
            ↪  Stack
42          void print(std::ostream& os, char ofc = ' ') const; //print elements
            ↪  of Stack to ostream os; this function prints elements in the
            ↪  opposite order, the oldest element should be printed first (last
            ↪  in last print)
43      };
44
45      /***** Overloading non-member global functions *****/
46      template <typename T, typename Container = std::deque<T>>
47      std::ostream& operator<<(std::ostream& os, const Stack<T, Container>& a);
        ↪  //invokes the print() method to print the Stack<T, Container> a in
        ↪  the specified ostream
48
49      template <typename T, typename Container = std::deque<T>>
50      bool operator==(const Stack<T, Container>& lhs, const Stack<T,
        ↪  Container>& rhs); // returns true if the two compared Stacks have the
        ↪  same elements, in the same order.
51
52      template <typename T, typename Container = std::deque<T>>
53      bool operator!=(const Stack<T, Container>& lhs, const Stack<T,
        ↪  Container>& rhs); // opposite of operator==()
54
55      template <typename T, typename Container = std::deque<T>>
56      bool operator<=(const Stack<T, Container>& lhs, const Stack<T,
        ↪  Container>& rhs); // returns true if every element in Stack lhs is
        ↪  smaller than or equal to the corresponding element of Statck rhs,
        ↪  until the end of lhs is reached
57
58      //include the implementation file
59      #include "stack.hpp"
60
```

```
}// end of namespace cop4530


#endif
```

4.2.2 Stack() [zero]

```
//use initialization list to call zero-parameter constructor of the
↪  internal Container
template <typename T, typename Container>
Stack<T, Container>::Stack() : c() {}
```

4.2.3 ~Stack()

```
//do nothing; destructor of data member (the container c) will be called
↪  automatically
template <typename T, typename Container>
Stack<T, Container>::~Stack() {}
```

4.2.4 Stack() [copy]

```
//utilize copy constructor of the internal container
template <typename T, typename Container>
Stack<T, Container>::Stack(const Stack<T, Container>& rhs) : c{rhs.c} {}
```

4.2.5 Stack() [move]

```
// utilize move constructor of the internal container
template <typename T, typename Container>
Stack<T, Container>::Stack(Stack<T, Container>&& rhs) : c
↪  {std::move(rhs).c} {}
```

4.2.6 operator=() [copy]

```
template <typename T, typename Container>
Stack<T, Container>& Stack<T, Container>::operator=(const Stack<T,
↪  Container>& rhs) {
  c = rhs.c; //apply copy assignment operator of the internal container
  return *this;
}
```

4.2.7 operator=() [move]

template <typename T, typename Container> Stack<T, Container>& Stack<T, Container>::operator=(S
Container>&& rhs) { c = std::move(rhs).c; return *this; }

4.2.8 empty()

```
template <typename T, typename Container>
bool Stack<T,Container>::empty() const {
  return c.empty();
}
```

4.2.9 clear()

template <typename T, typename Container> void Stack<T,Container>::clear() { c.clear(); }

4.2.10 push()

Copy version:

```
template <typename T, typename Container>
void Stack<T,Container>::push(const T& x) {
  c.push_back(x);
}
```

4.2.11 pop()

```
template <typename T, typename Container>
void Stack<T,Container>::pop() {
  c.pop_back();
}
```

4.2.12 top()

```
//const version
template <typename T, typename Container>
const T& Stack<T,Container>::top() const {
  return c.back();
}
```

4.2.13 top() reference version

```
template <typename T, typename Container>
T& Stack<T,Container>::top() {
  return c.back();
}
```

4.2.14 size()

```
template <typename T, typename Container>
int Stack<T,Container>::size() const {
  return c.size();
}
```

4.2.15 print()

```
template <typename T, typename Container>
void Stack<T,Container>::print(std::ostream& os, char ofc) const {
  //check if the container is empty
  if (empty())
    return;
  //print first element (the oldest element)
  os << c.front();

  //print the rest of elements, if there is any
  for (auto itr = c.begin() + 1; itr != c.end(); ++itr) {
    os << ofc << *itr;
  }
}
```

4.2.16 Non-member Global functions

For the following functions, we'll implement in different ways. `operator«()` will not require access to `Stack`'s private/protected member, it only calls the `print()` routine from the `Stack` class. For other three operators (`==`, `!=` and `<=`), we declare them as friend of `Stack` class, so they can access private/protected members of `Stack` class. (it is also possible to define a public function in `Stack`, and call by these operators).

4.2.16.1 operator«()

```
template <typename T, typename Container>
std::ostream& operator<<(std::ostream& os, const Stack<T,Container>& a) {
  a.print(os);
  return os;
}
```

4.2.16.2 operator==()

In header of `Stack`, we declare this function as the friend of `Stack`:

```
template <typename A, typename B>
friend bool operator==(const Stack<A, B>& lhs, const Stack<A, B>& rhs);
```

Pay attention that, you are declaring a function template as the friend of `Stack`, so you must provide full description of the header of the function template.

After this, we can implement the function template outside of `Stack` class:

```
template <typename T, typename Container>
bool operator==(const Stack<T,Container>& lhs, const Stack<T,Container>&
 ↪  rhs) {
  //check size first
  if (lhs.size() != rhs.size())
    return false;

  for(auto itr_l = lhs.c.begin(), itr_r = rhs.c.begin(); itr_l !=
   ↪  lhs.c.end(); ++itr_l, ++itr_r) {
    if (*itr_l != *itr_r)
      return false;
  }

  return true;
}
```

4.2.16.3 `operator!=()`

We will call `operator==()` to finish the work.

```
template <typename T, typename Container>
bool operator!=(const Stack<T,Container>& lhs, const Stack<T,Container>&
 ↪  rhs) {
  return !(lhs == rhs);
}
```

4.2.16.4 `operator<=()`

Similarly, we declare this function template as friend of `Stack`:

```
template <typename A, typename B>
friend bool operator<=(const Stack<A, B>& lhs, const Stack<A, B>& rhs);
```

Then, we implement it outside of `Stack`:

```
template <typename T, typename Container>
bool operator<=(const Stack<T,Container>& lhs, const Stack<T,Container>&
 ↪  rhs) {
  //test if the size of lhs is larger than rhs, if so, return false
  if(lhs.size() > rhs.size())
    return false;

```

```
7     for(auto itr_l = lhs.c.begin(), itr_r = rhs.c.begin(); itr_l !=
      ↪  lhs.c.end(); ++itr_l, ++itr_r) {
8       //check if the current entry in lhs satisfies the condition
9       if (*itr_l > *itr_r)
10        return false;
11    }
12
13    return true;
14
15  }
```

Chapter 5

Queue

Chapter 6

Tree

6.1 Binary Search Tree

6.1.1 General Idea

A binary search tree is a binary tree implemented with the following rule: a node's left child is no larger than the node; a node's right child is no smaller than the node; Under this rule, it is clear that the smallest node in binary search tree is the leftmost node, while the largest node is the rightmost node.

6.1.2 Simple Implementation

In this section, a simple binary search tree template will be implemented. The header of the class would be:

```
template <typename comparable>
class BinarySearchTree {

};
```

`comparable` is the name of a class that supports comparison by `operator<()`. Since the building of binary search tree requires ordering of nodes.

The header file of binary search tree class template is as follows:

```
1  #pragma once
2  #include <iostream>
3
4  template <typename comparable>
5  class BinarySearchTree {
6  private:
7    //nested tree node structure
8    struct BinaryNode {};
```

```
9
10 private:
11   BinaryNode* root;
12
13 private:
14   /** private operating functions **/
15   ///insert
16   void insert(const comparable& val, BinaryNode* & t); //copy
17   void insert(comparable&& val, BinaryNode* & t); //move
18
19   ///remove
20   void remove(const comparable& val, BinaryNode* & t);
21
22   ///search
23   BinaryNode* findMin(BinaryNode* t) const;
24   BinaryNode* findMax(BinaryNode* t) const;
25   bool contains(const comparable& val, BinaryNode* t) const;
26
27   ///utility
28   void makeEmpty(BinaryNode* & t);
29   void printTree(BinaryNode* t, std::ostream& out) const;
30   BinaryNode* clone(BinaryNode* t) const;
31
32 public:
33   /** Constructor and destructor **/
34   BinarySearchTree(); //zero-parameter default constructor
35   BinarySearchTree(const BinarySearchTree& rhs); //copy constructor
36   BinarySearchTree(BinarySearchTree&& rhs); //move constructor
37   ~BinarySearchTree(); //destructor
38
39   /** Assignment operator **/
40   BinarySearchTree& operator=(const BinarySearchTree& rhs); //copy
41   BinarySearchTree& operator=(BinarySearchTree&& rhs); //move
42
43   /** Public Search Interface **/
44   const comparable& findMin() const;
45   const comparable& findMax() const;
46   bool contains(const comparable& val) const;
47
48   /** Modification of tree **/
49   void makeEmpty();
50   void insert(const comparable& val);//copy version
51   void insert(comparable&& val);//move version
52   void remove(const comparable& val);
53
```

```
54    /** Utility **/
55    bool isEmpty() const;
56    void printTree(std::ostream& out = std::cout) const;
57
58  };
59
60  //include implementation here
61  #include "bst.hpp"
```

An object of `BinarySearchTree` class holds a private data member `root`, which is a pointer to `TreeNode` type, holds the address of the root of a binary tree. The implementation of `TreeNode` structure and other member functions are detailed below.

6.1.2.1 Tree node structure

A tree node contains three data members, one is `comparable` type and is used to hold the data of that node. The other two are pointer to tree node, which will be used to hold the address of the node's left and right child. Code:

```
1  struct BinaryNode {
2    comparable element;//data stored in the node, and it is comparable (at
     ↪  least one comparable routine is defined for this type)
3    BinaryNode* left;
4    BinaryNode* right;
5
6    ///constructor
7    //copy
8    BinaryNode(const comparable& val = val{}, BinaryNode* lt = nullptr,
     ↪  BinaryNode* rt = nullptr) : element {val}, left {lt}, right {rt} {}
9    //move
10   BinaryNode(comparable&& val = val{}, BinaryNode* lt = nullptr,
     ↪  BinaryNode* rt = nullptr) : element {std::move(val)}, left {lt},
     ↪  right {rt} {}
11 };
```

6.1.2.2 `zero parameter constructor`

Just initialize `root` pointer as `nullptr`. Code:

```
1  template <typename comparable>
2  BinarySearchTree<comparable>::BinarySearchTree() : root {nullptr} {}
```

6.1.2.3 `copy constructor`

This constructor accepts another object of `BinarySearchTree` class. It will call an internal recursive routine `clone()` to finish copying and building. The details of the process is in `clone()` function. Code:

```
template <typename comparable>
BinarySearchTree<comparable>::BinarySearchTree(const BinarySearchTree& rhs)
  ↪ {
  root = clone(rhs.root);
}
```

6.1.2.4 move constructor

It is very simple to move, we just need to "bring" rhs's root to our root, and redirect `rhs.root` to `nullptr`. Code:

```
template <typename comparable>
BinarySearchTree<comparable>::BinarySearchTree(BinarySearchTree&& rhs) {
  root = rhs.root;
  rhs.root = nullptr;
}
```

6.1.2.5 destructor

Just call `makeEmpty()` routine, all memories will be recycled. Code:

```
template <typename comparable>
BinarySearchTree<comparable>::~BinarySearchTree() {
  makeEmpty();
}
```

6.1.2.6 copy assignment operator

We call `clone()` to do the copy and building work. Since this is assignment operator, don't forget return value. Code:

```
template <typename comparable>
BinarySearchTree<comparable>& BinarySearchTree<comparable>::operator=(const
  ↪ BinarySearchTree& rhs) {
  root = clone(rhs.root);
  return *this;
}
```

6.1.2.7 move assignment operator

Take `rhs`'s root directly. Remember the return. Code:

```
template <typename comparable>
BinarySearchTree<comparable>&
  ↪ BinarySearchTree<comparable>::operator=(BinarySearchTree&& rhs) {
  root = rhs.root;
  rhs.root = nullptr;
```

```
5   return *this;
6 }
```

6.1.2.8 public findMin()

This function is in public domain. It will return the constant reference of the minimum node in the tree. Like many other public member functions, this function will call a private version of findMin() to actually finish the job. This is because of the recursive nature of the tree data structure, many functions will work recursively. Code:

```
1 template <typename comparable>
2 const comparable& BinarySearchTree<comparable>findMin() const {
3   return (findMin(root))->element;
4 }
```

6.1.2.9 public findMax()

Similar with findMin(). Code:

```
1 template <typename comparable>
2 const comparable& BinarySearchTree<comparable>::findMax() const {
3   return (findMax(root))->element;
4 }
```

6.1.2.10 public contains()

This function accepts a parameter val of type comparable. It will search the tree for the existence of val. It will call a private recursive version of contains(). Code:

```
1 template <typename comparable>
2 bool BinarySearchTree<comparable>::contains(const comparable& val) const {
3   return contains(val, root);
4 }
```

6.1.2.11 public makeEmpty()

This function will clear all nodes (reclaim their memory) in the tree. It will call a private recursive version of makeEmpty(). Code:

```
1 template <typename comparable>
2 void BinarySearchTree<comparable>::makeEmpty() {
3   makeEmpty(root);
4 }
```

6.1.2.12 public insert()

This function accepts a parameter val of type comparable. It will call a private recursive version of insert() to insert val into the tree (to the proper position where val should go).

Code (copy version):

```
template <typename comparable>
void BinarySearchTree<comparable>::insert(const comparable& val) {
  insert(val, root);
}
```

6.1.2.13 public remove()

This function accepts a parameter `val` of type `comparable`. It will call a private recursive version of `remove()` to remove `val` from the tree. Code:

```
template <typename comparable>
void BinarySearchTree<comparable>::remove(const comparable& val) {
  remove(val, root);
}
```

6.1.2.14 public isEmpty()

This function will check if the tree is empty. The criteria is simple: if the root is `nullptr`, then the tree is empty. Code:

```
template <typename comparable>
bool BinarySearchTree<comparable>::isEmpty() const {
  if (root == nullptr)
    return true;
  else
    return false;
}
```

6.1.2.15 public printTree()

This function accepts a `std::ostream` object `out`. It will call a private version of `printTree()` and pass this object into it, to print the tree in in-order (in ascending order). Code:

```
template <typename comparable>
void BinarySearchTree<comparable>::printTree(std::ostream& out) const {
  printTree(root, out);
}
```

6.1.2.16 insert()

The private recursive version of `insert()`. Implemented recursively. It accepts two parameters: a `comparable` type `val`, and a pointer to `TreeNode` type `t`. The function can insert the value into the subtree whose root is indicated by `t`. Its working steps are:

- check if `t` is pointing to `nullptr`, if so, this is the base case: an empty branch is found, and `val` should be inserted there;

- if it is not the base case, we will insert it into `t`'s children:
  - `val > t->element`: insert to right subtree by calling itself and pass `val` and `t->right`
  - `val < t->element`: insert to left subtree by calling itself and pass `val` and `t->right`
- if `val == t->element`, we do nothing, since its already in the tree (no duplicate)

Code (copy version):

```
template <typename comparable>
void BinarySearchTree<comparable>::insert(const comparable& val,
↪  BinaryNode* & t) {
  //base case1: t is pointing to nullptr
  if (t == nullptr) {
    t = new BinaryNode{val}; //this step will modify t, so pass the pointer
    ↪  by reference is necessary
    return;
  }

  //determine which branch to insert
  //not considering the equal case
  if (val < t->element)
    insert(val, t->left);
  else if (val > t->element)
    insert(val, t->right);
  else
    return;//val == t->element, do nothing
}
```

Notice that the passed in `TreeNode` pointer type is referenced type, this is because we will change the memory address stored in pointer itself when we allocate new chunk of memory and store the new tree node.

6.1.2.17 `remove()`

This is internal private version of `remove()`. It accepts two parameters: a `comparable` type `val`, a reference to pointer of `TreeNode` type (we need to change the address stored in pointer, so we need reference type pointer). This function works recursively. It will remove the node containing `val` in subtree whose root is pointed by `t`. When we remove a node from the tree, its children are disconnected from the tree (because this node connects them to the tree). We need to reconnect them to the tree. The details of reconnecting protocol is up to programmer's choice, here we'll introduce a simple way.

- there are four base cases
  1. `t == nullptr`: no match found, return

2. `t->element > val`: call `remove(val, t->left)`
3. `t->element < val`: call `remove(val, t->right)`
4. `t->element == val`: this is the node we want to remove, proceed to next step

- find the left most leaf of `t->right`: `left_leaf_ptr`
- attach `t->left` to the left child of `left_leaf_ptr`
- use a temporary `TreeNode` pointer `temp` to store address of `t->right`
- reclaim `t`'s memory
- reconnect previous `t`'s children by: `t = temp`. Notice that `t` should point to its parent's children. Before deletion, `t`'s parent's child is `t`, now, `t`'s parent's child is `t->right`, `t->left` is also connected to `t->right`.

Code:

```
template <typename comparable>
void BinarySearchTree<comparable>::remove(const comparable& val,
↪  BinaryNode* & t) {
  //base case 1: t is pointing to nullptr, no match
  if (t == nullptr)
    return;

  //base case 2: t is pointing to the target node
  if (t->element == val) {
    //find the left most spot of t->right, and attach t->left to it
    if (t->right == nullptr) {
      t->right = t->left;
    }

    else {
      BinaryNode* left_leaf_ptr = t->right;
      while (left_leaf_ptr -> left != nullptr)
        left_leaf_ptr = left_leaf_ptr -> left;
      //after the above loop, left_leaf_ptr is pointing to the left most
      ↪  leaf of t->right, attach t->left to the left subtree of this leaf
      left_leaf_ptr -> left = t->left;
    }

    //keep record of the address of current t->right
    BinaryNode* temp = t->right;
    //reclaim memory
    delete t;
    //re-connect tree node
    t = temp; // here requires modifying t, thus reference is required
```

```
28
29      return;
30    }
31
32    //t is not pointing to the target node
33    if (t->element > val)
34      remove(val, t->left);
35    else
36      remove(val, t->right);
37  }
```

6.1.2.18 findMin()

This function will return a pointer of `TreeNode` type which points to the left most leaf of the tree whose root is pointed by the passed in `TreeNode` pointer `t`. If `t == nullptr`, `nullptr` will be returned. Code:

```
1  template <typename comparable>
2  typename BinarySearchTree<comparable>::BinaryNode*
   ↪  BinarySearchTree<comparable>::findMin(BinaryNode* t) const {
3    if (t == nullptr)
4      return t;
5
6    while (t->left != nullptr)
7      t = t->left;
8    //after the above loop, t is now pointing to left-most leaf
9    return t;
10 }
```

Pay attention to the return type keyword:

```
typename BinarySearchTree<comparable>::BinaryNode*
```

If you are returning a nested class type, for example, in the above code you are returning a pointer to `BinaryNode`, which itself is a structure defined in `BinarySearchTree`, you have to add the keyword `typename` to indicate this is a type to be returned.

6.1.2.19 findMax()

Similar with `findMin()`, this function will return a pointer to the right most node. Code:

```
1  template <typename comparable>
2  typename BinarySearchTree<comparable>::BinaryNode*
   ↪  BinarySearchTree<comparable>::findMax(BinaryNode* t) const {
3    if (t == nullptr)
4      return t;
5
6    while (t->right != nullptr)
```

```
    t = t->right;
  //after the above loop, t is not pointing to right-most leaf
  return t;
}
```

6.1.2.20 contains()

Steps to find a specific node is similar with `remove()`. Code:

```
template <typename comparable>
bool BinarySearchTree<comparable>::contains(const comparable& val,
↪  BinaryNode* t) const {
  //base case: t == nullptr, no match found
  if (t == nullptr)
    return false;

  //base case2: t->element == val
  if (t->element == val)
    return true;

  //try to find val in t's children
  if (t->element > val)
    return contains(val, t->left);
  else
    return contains(val, t->right);
}
```

6.1.2.21 makeEmpty()

This function accepts a pointer of `TreeNode` type `t`. It will reclaim all memory used by this node and all its children. Working steps:

- check if base case reached (`t == nullptr`), if so, do nothing, return
- call itself and pass `t->left` to reclaim memory of its left child
- call itself and pass `t->right` to reclaim memory of its right child
- reclaim `t`'s memory, and assign it to `nullptr`

Code:

```
template <typename comparable>
void BinarySearchTree<comparable>::makeEmpty(BinaryNode* & t) {
  //base case
  if (t == nullptr)
    return;

  //begin makeEmpty
```

```
8   makeEmpty(t->left);
9   makeEmpty(t->right);
10  delete t;
11  t = nullptr;
12 }
```

6.1.2.22 printTree()

The idea is similar with `makeEmpty()`, the only difference is in `printTree`, you are printing rather than deleting. Code:

```
1  template <typename comparable>
2  void BinarySearchTree<comparable>::printTree(BinaryNode* t, std::ostream&
   ↪  out) const {
3    //base case
4    if (t == nullptr)
5      return;
6
7    //print the tree in inorder traversal
8    printTree(t->left, out);
9    out << t->element << ' ';
10   printTree(t->right, out);
11 }
```

6.1.2.23 clone()

This function accepts a pointer to `TreeNode` type `t`. It will return a pointer to a newly constructed `TreeNode`, whose element is the same as `t->element`, left child is the same as `t->left`, right child is the same as `t->right`. It works in a recursive way. Code:

```
1  typename BinarySearchTree<comparable>::BinaryNode*
   ↪  BinarySearchTree<comparable>::clone(BinaryNode* t) const {
2    /** pay attention that what you clone is a BinaryNode! **/
3    //base case
4    if (t == nullptr)
5      return t;
6
7    //clone
8    return new BinaryNode{t->element, clone(t->left), clone(t->right)};
9  }
```

Pay attention that, the returned pointer is constructed by the address generated by the `new` operation (allocating new memory spaces).

6.1.3 Problem with Simple Binary Search Tree

The binary search tree can only guarantee $O(\log N)$ complexity when the tree is nearly **BALANCED**, which means for any node in the binary search tree, the number of nodes in its left subtree is roughly the same as its right subtree. However, this may not be the case during practical uses of this simple binary search tree. Consider two cases:

1. We insert an ordered array into the tree by calling `insert()` repeatedly for all the elements in array in order. We'll create a linked list rather than a binary tree. If its in ascending order, only right subtree will be used; If its in descending order, only left subtree will be used. Many operations will be $O(N)$ complexity.
2. We have a balanced binary search tree at first. We kept removing nodes in it by calling `remove()`. In our implementation of `remove()`, we will attach the target node's left subtree to its right subtree. So this will decrease the number of nodes in left subtrees. The balanced tree will degenerate to un-balanced tree, with one subtree holds significant more amount of nodes than the other subtree.

In both cases, we may face increased time complexity. Thus, we want to come up with ways to build **balanced** binary search tree.

6.2 AVL Tree

6.2.1 General Idea

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by most 1 (the height of an empty tree is defined to be -1).

Let $S(h)$ be the minimum number of nodes that an AVL tree of height h needs. Then we have: $S(h) = S(h-1) + S(h-2) + 1$, where $S(h-1)$ is the number of nodes of the higher child of `root`, $S(h-2)$ is the number of nodes of the lower child of `root`, 1 corresponds to one layer from child to `root`.

6.2.2 AVL-property Loss and Fix

In this section, we'll talk about the cause of AVL-property loss and ways to fix it. Both insertion and deletion operation can violate AVL-property, since they can bring height change to the tree. We'll analyze them one by one.

6.2.2.1 Insertion caused unbalance and fix

In Figure 6.1, before insertion, N's left and right subtree's height was already differred by 1 (dashed line indicates possible insertion site). After a node is inserted into one of its subtree (LL, LR, RL, RR), the height of N's left and right subtree is now differred by two, AVL property lost for node N. There are four possible cases:

1. Insert into `N->left->left` (insert into LL)

2. Insert into `N->left->right` (insert into LR)
3. Insert into `N->right->left` (insert into RL)
4. Insert into `N->right->right` (insert into RR)

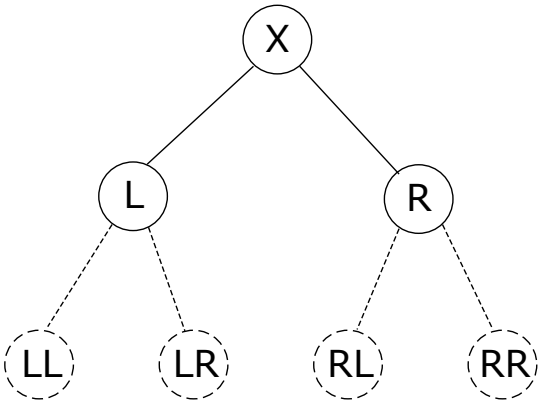

Figure 6.1: AVL tree illustration

Case 1 and case 4 are illustrated in a greater detail in Figure 6.2. Case 2 and case 3 are illustrated in a greater detail in Figure 6.3.

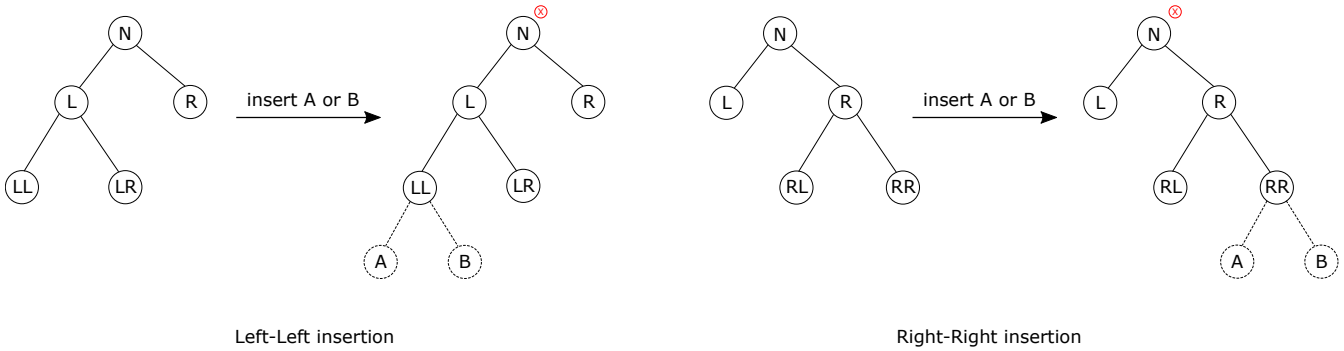

Figure 6.2: Left-left insertion and right-right insertion

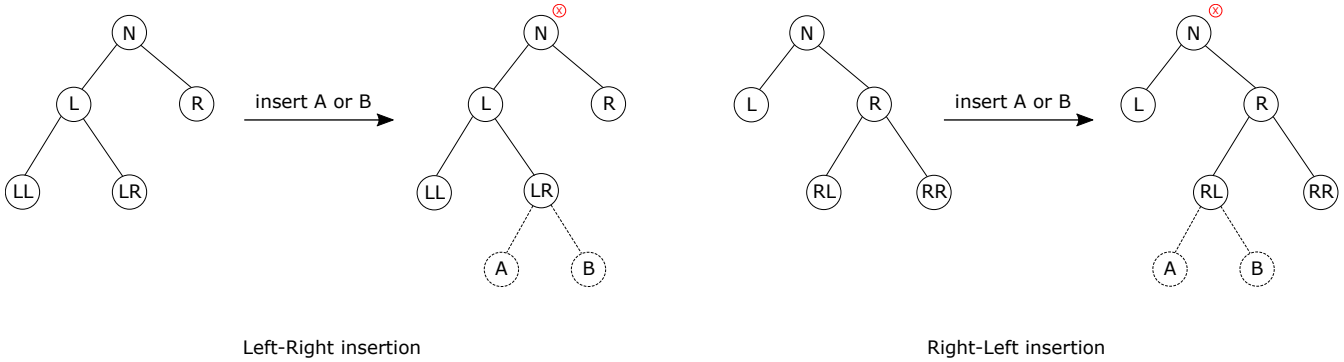

Figure 6.3: Left-right insertion and right-left insertion

One thing should be pointed out is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their AVL-property lost, because only those nodes have their subtrees altered. And we only need to fix the first node who lost AVL-property, so the rest nodes above it can restore their AVL-property.

To fix AVL-property loss by LL-insertion and RR-insertion, we can perform a single rotation. Specifically:

- to fix LL-insertion, we rotate node `N` with its left child. In this process, `N` will become `N->left`'s right child, while `N->left`'s right child will become `N`'s left child.
- to fix RR-insertion, we rotate node `N` with its right child. In this process, `N` will become `N->right`'s left child, while `N->right`'s left child will become `N`'s right child.

The process is illustrated in Figure 6.4. The letters in node represents the original position of the node.

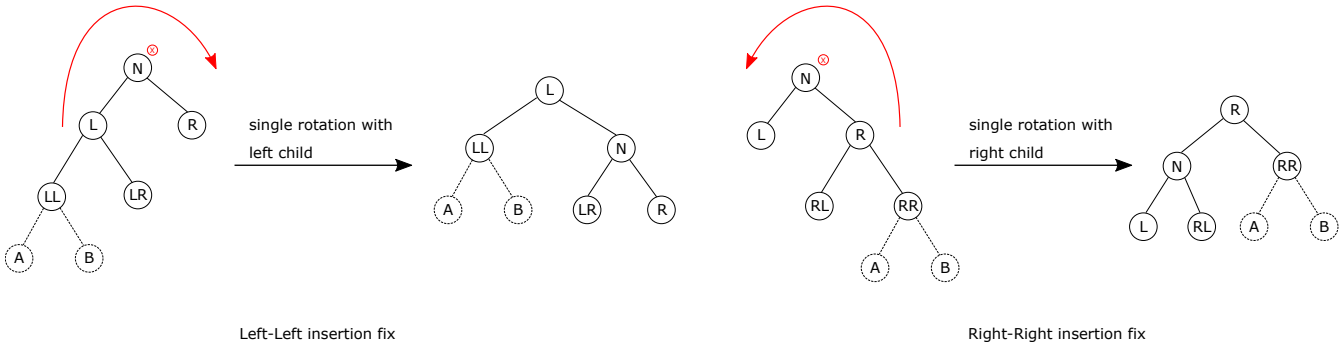

Figure 6.4: Single rotation to fix AVL-property loss caused by Left-left insertion and right-right insertion

From Figure 6.4, we can see that single rotation can modify the height of node `LL` or `RR`, no matter `A` or `B` is inserted.

To fix AVL-property loss by LR-insertion and RL-insertion, we have to perform two rotations. Specifically:

- to fix LR-insertion
  - rotate `N->left` with `N->left`'s right child (single rotation)
  - rotate `N` with `N`'s left child (single rotation)
- to fix RL-insertion
  - rotate `N->right` with `N->right`'s left child (single rotation)
  - rotate `N` with `N`'s right child (single rotation)

The process to fix LR-insertion is illustrated in Figure 6.5. The process to fix RL-insertion is illustrated in Figure 6.6.

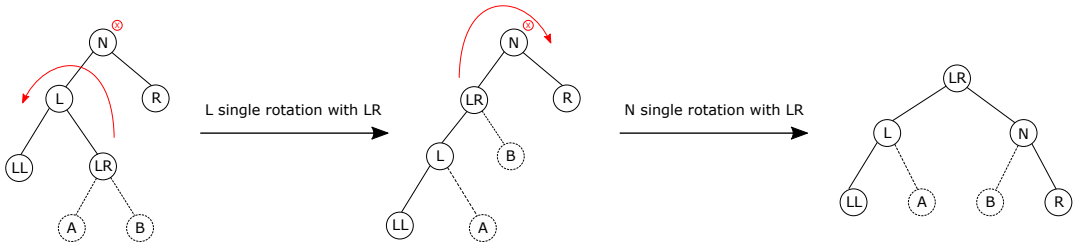

Figure 6.5: Double rotation to fix AVL-property loss caused by Left-right insertion

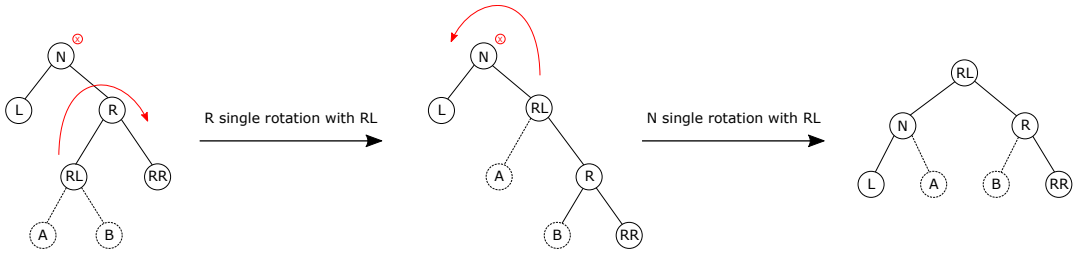

Figure 6.6: Double rotation to fix AVL-property loss caused by Right-left insertion

6.2.2.2 Deletion caused unbalance and fix

There are four cases of AVL-property loss by deletion. They are shown in Figure 6.7. The dashed node E indicates where deletion occured. Node E represent the only child of its parent node (doesn't matter if its left or right child). After deleting, its parent's height will be reduced by 1.

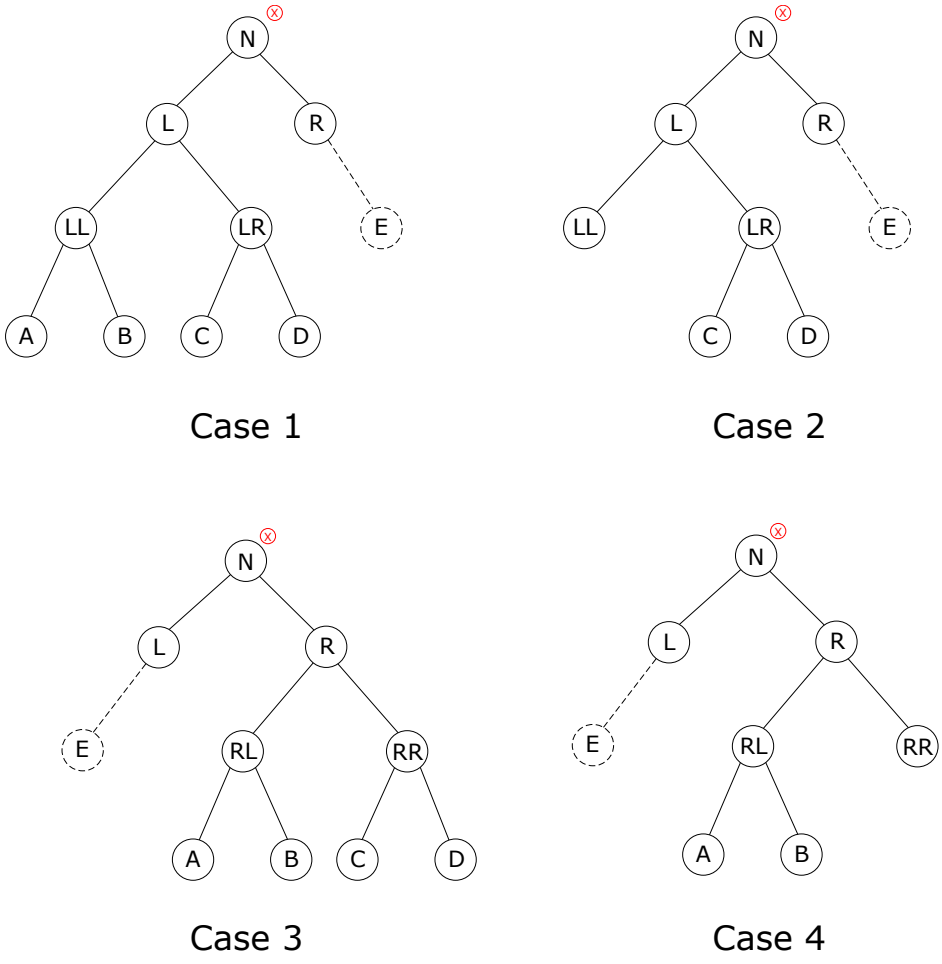

Figure 6.7: AVL-property loss by deletion

To conclude the four cases:

Case 1: `height(N->left) > height(N->right) + 1`

`&& height(N->left->left) == height(N->left->right)`

Case 2: `height(N->left) > height(N->right) + 1`

`&& height(N->left->left) < height(N->left->right)`

Case 3: `height(N->right) > height(N->left) + 1`

`&& height(N->right->right) == height(N->right->left)`

Case 4: `height(N->right) > height(N->left) + 1`

`&& height(N->right->right) < height(N->right->left)`

Take a closer look, case 1 is similar with LL-insertion case, because they both can be solved with a single rotation of `N` with `N->left`. This is because, a single rotation of `N` with `N->left` can lower node `R` one layer, so the height difference between `R` and bottom nodes will reduced from 2 to 1. Although it can also rise node `LL` one layer, it will not violate AVL-

property since `height(LL) == height(LR)` before single rotation. After the single rotation, `heigh(LL)` and `height(LR)` differ by 1, which is within the requirement. However, this may not be the case when `height(LL) < height(LR)` initially. Because single rotation will cause `height(LL) < height(LR) + 1`. This is actually case 2, which is similar with LR-insertion case. They both can be solved with a double rotation: first rotate `L` with `L->right`, then rotate `N` with `N->left`. The double rotation will rise the height of both node `C` and `D`, and will lower the heigh of `R` by 1 (first single rotation: rise `D`, lower `LL`; second single rotation: rise `LL` and `C`, lower `R`).

Using same argument, case 3 is similar with RR-insertion case, and can be fixed by a single rotation of `N` with `N->right`. Case 4 is similar with RL-insertion, and can be fixed by a double rotation: first rotate `R` with `R->left`, then rotate `N` with `N->right`. The fixing for case 2 is the same shown in Figure 6.5. The fixing for case 4 is the same shown in Figure 6.6. The fixing for case 1 and case 3 is similar with 6.4, they are drawn one more time in Figure 6.8 and Figure 6.9.

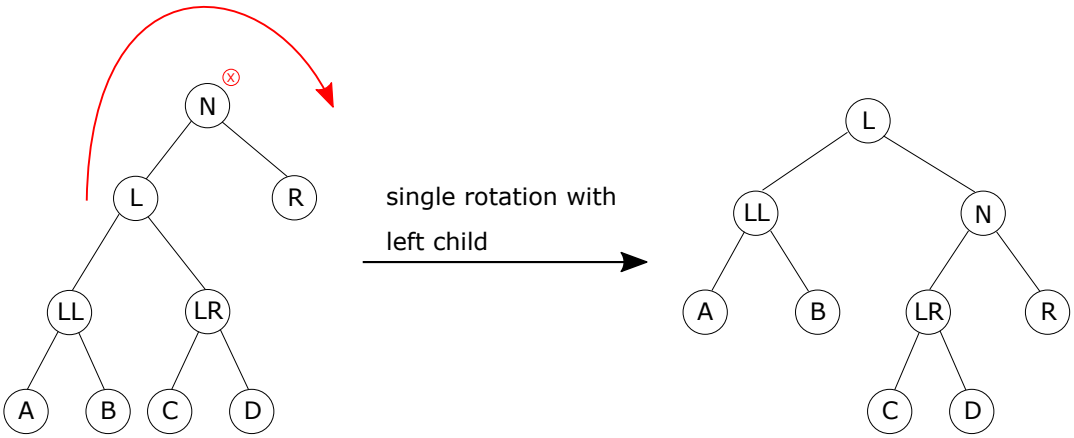

Figure 6.8: Balancing AVL tree. When left subtree is too high and its two branches have same height

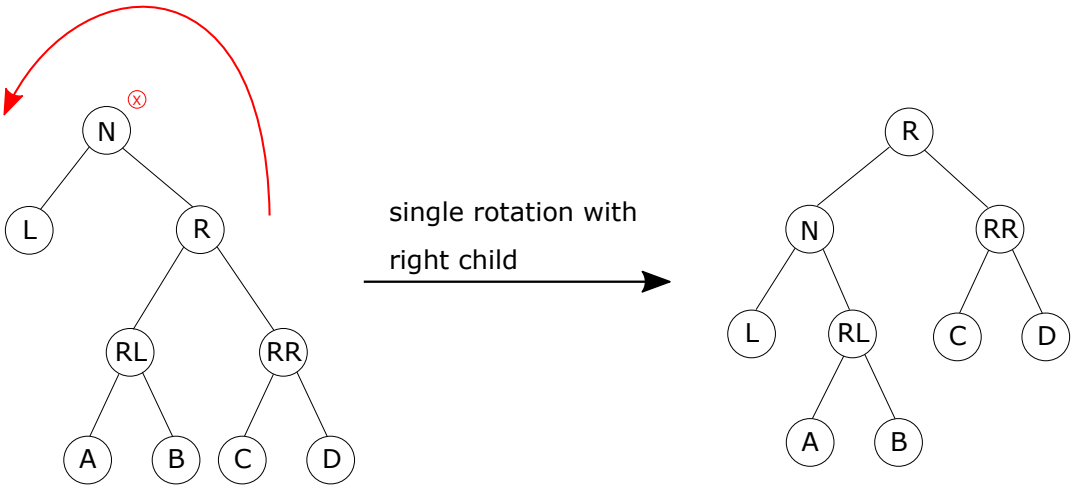

Figure 6.9: Balancing AVL tree. When right subtree is too high and its two branches have same height

6.2.2.3 Summary

From the above analyze, we can see that, if we want to restore the balance for a node `N`, we first check which side is **TOO** high (which means the height difference is larger than 1). Then we check if the **inner** branch of this side is higher than the **outter** branch, if so, we use double rotation to fix (only double rotation can fix the issue caused by higher inner branch), otherwise, we use single rotation (single rotation is enough to fix issue caused by higher outter branch).

6.2.3 Simple Implementation (recursive)

In this section, a simple implementation of AVL tree is discussed. It uses recursive algorithm. Unlike ordinary binary search tree, we have to store extra information in each tree node, which is the height of the node.

6.2.3.1 header

We define a nested structure `AvlNode` inside the `AVLTree` class. The data member of `AVLTree` class is just a pointer to `AvlNode` type, which will be used to hold the root of the AVL tree. We also define a static constant integer variable to indicate the maximum allowed imbalance:

```
static const int ALLOWED_IMBALANCE = 1;
```

Similar with binary tree, we'll use recursive function to implement the AVL tree. It may have slower performance, but is easier to read and understand. What is unique about AVL tree is that we keep it balanced, which is done by the following routines:

```
1  void balance(AvlNode* &t); //internal balance routine
2  void rotateWithLeftChild(AvlNode* &t);
3  void rotateWithRightChild(AvlNode* &t);
4  void doubleWithLeftChild(AvlNode* &t);
5  void doubleWithRightChild(AvlNode* &t);
```

We'll discuss each later. The code for the whole header file is as follows:

```
1  #pragma once
2  #include <iostream>
3  #include <algorithm>
4
5  template <typename comparable>
6  class AVLTree {
7  private:
8    //nested tree node structure
9    struct AvlNode { // defined later
10   };
11
12 private:
13   AvlNode* root;
```

```
14    static const int ALLOWED_IMBALANCE = 1;
15
16  private:
17    /** private operating functions **/
18    ///insert
19    void insert(const comparable& val, AvlNode* & t); //copy
20    void insert(comparable&& val, AvlNode* & t); //move
21
22    ///remove
23    void remove(const comparable& val, AvlNode* & t);
24
25    ///search
26    AvlNode* findMin(AvlNode* t) const;
27    AvlNode* findMax(AvlNode* t) const;
28    bool contains(const comparable& val, AvlNode* t) const;
29
30    ///utility
31    void makeEmpty(AvlNode* & t);
32    void printTree(AvlNode* t, std::ostream& out) const;
33    AvlNode* clone(AvlNode* t) const;
34    int height(AvlNode* t) const;
35    void balance(AvlNode* &t); //internal balance routine
36    void rotateWithLeftChild(AvlNode* &t);
37    void rotateWithRightChild(AvlNode* &t);
38    void doubleWithLeftChild(AvlNode* &t);
39    void doubleWithRightChild(AvlNode* &t);
40
41  public:
42    /** Constructor and destructor **/
43    AVLTree(); //zero-parameter default constructor
44    AVLTree(const AVLTree& rhs); //copy constructor
45    AVLTree(AVLTree&& rhs); //move constructor
46    ~AVLTree(); //destructor
47
48    /** Assignment operator **/
49    AVLTree& operator=(const AVLTree& rhs); //copy
50    AVLTree& operator=(AVLTree&& rhs); //move
51
52    /** Public Search Interface **/
53    const comparable& findMin() const;
54    const comparable& findMax() const;
55    bool contains(const comparable& val) const;
56
57    /** Modification of tree **/
58    void makeEmpty();
```

```
59    void insert(const comparable& val);//copy version
60    void insert(comparable&& val);//move version
61    void remove(const comparable& val);
62
63    /** Utility **/
64    bool isEmpty() const;
65    void printTree(std::ostream& out = std::cout) const;
66
67  };
68
69  //include implementation here
70  #include "avl.hpp"
```

6.2.3.2 AVL node structure

The node of AVL tree uses a variable to hold the height information of the node. During construction, the default height is 0, the constructor accepts an integer to pass it to the height member, setting the height during construction. Code:

```
1  struct AvlNode {
2    comparable element;//data stored in the node, and it is comparable (at
     ↪  least one comparable routine is defined for this type)
3    AvlNode* left;
4    AvlNode* right;
5    int height; //store the height of this node
6
7    ///constructor
8    //copy
9    AvlNode(const comparable& val = val{}, AvlNode* lt = nullptr, AvlNode* rt
     ↪  = nullptr, int h = 0) : element {val}, left {lt}, right {rt}, height
     ↪  {h} {}
10   //move
11   AvlNode(comparable&& val = val{}, AvlNode* lt = nullptr, AvlNode* rt =
     ↪  nullptr, int h = 0) : element {std::move(val)}, left {lt}, right
     ↪  {rt}, height {h} {}
12 };
```

6.2.3.3 `zero parameter constructor`

`root` will be initialized to `nullptr`. Code:

```
1  template <typename comparable>
2  AVLTree<comparable>::AVLTree() : root {nullptr} {}
```

6.2.3.4 `copy constructor`

`clone()` routine will be used, similar with binary search tree. Code:

```
template <typename comparable>
AVLTree<comparable>::AVLTree(const AVLTree& rhs) {
  root = clone(rhs.root);
}
```

6.2.3.5 move constructor

Same as binary search tree, code:

```
template <typename comparable>
AVLTree<comparable>::AVLTree(AVLTree&& rhs) {
  root = rhs.root;
  rhs.root = nullptr;
}
```

6.2.3.6 destructor

Same as binary search tree, code:

```
template <typename comparable>
AVLTree<comparable>::~AVLTree() {
  makeEmpty();
}
```

6.2.3.7 copy assignment operator

Same as binary search tree, code:

```
AVLTree<comparable>& AVLTree<comparable>::operator=(const AVLTree& rhs) {
  root = clone(rhs.root);
  return *this;
}
```

6.2.3.8 move assignment operator

Same as binary search tree, code:

```
template <typename comparable>
AVLTree<comparable>& AVLTree<comparable>::operator=(AVLTree&& rhs) {
  root = rhs.root;
  rhs.root = nullptr;
  return *this;
}
```

6.2.3.9 public findMin()

Same as binary search tree, code:

```
template <typename comparable>
const comparable& AVLTree<comparable>::findMin() const {
  return (findMin(root))->element;
}
```

6.2.3.10 public findMax()

Same as binary search tree, code:

```
template <typename comparable>
const comparable& AVLTree<comparable>::findMax() const {
  return (findMax(root))->element;
}
```

6.2.3.11 public contains()

```
template <typename comparable>
bool AVLTree<comparable>::contains(const comparable& val) const {
  return contains(val, root);
}
```

6.2.3.12 public makeEmpty()

```
template <typename comparable>
void AVLTree<comparable>::makeEmpty() {
  makeEmpty(root);
}
```

6.2.3.13 public insert()

Copy version:

```
template <typename comparable>
void AVLTree<comparable>::insert(const comparable& val) {
  insert(val, root);
}
```

6.2.3.14 public remove()

```
template <typename comparable>
void AVLTree<comparable>::remove(const comparable& val) {
  remove(val, root);
}
```

6.2.3.15 public isEmpty()

```
template <typename comparable>
bool AVLTree<comparable>::isEmpty() const {
```

```
3   if (root == nullptr)
4     return true;
5   else
6     return false;
7 }
```

6.2.3.16 public printTree()

```
1 template <typename comparable>
2 void AVLTree<comparable>::printTree(std::ostream& out) const {
3   printTree(root, out);
4 }
```

6.2.3.17 insert()

This function accepts a pointer to `AvlNode` type `t` and a `AvlNode` type variable `val`. It will try to insert `val` into subtree started with `t`. The inserting process is the same as binary search tree. After inserting, `balance()` will be called to:

1. check if balance of `t` is violated
2. balance the tree if necessary
3. update the height of the node pointed by `t`

Code (copy version):

```
1 template <typename comparable>
2 void AVLTree<comparable>::insert(const comparable& val, AvlNode* & t) {
3   //base case1: t is pointing to nullptr
4   if (t == nullptr) {
5     t = new AvlNode{val}; //this step will modify t, so pass the pointer by
      ↪  reference is necessary
6   }
7
8   //determine which branch to insert
9   //when item being inserted is the same as t->element, do nothing (don't
    ↪  insert)
10  else if (val < t->element)
11    insert(val, t->left);
12  else if (val > t->element)
13    insert(val, t->right);
14
15  balance(t);
16 }
```

6.2.3.18 `remove()`

Similar with `insert()`, `balance(t)` will be called after removing the node (which may cause potential height change). Code:

```
template <typename comparable>
void AVLTree<comparable>::remove(const comparable& val, AvlNode* & t) {
  //base case 1: t is pointing to nullptr, no match
  if (t == nullptr)
    return;

  //base case 2: t is pointing to the target node
  if (t->element == val) {
    //find the left most spot of t->right, and attach t->left to it
    if (t->right == nullptr) {
      t->right = t->left;
    }

    else {
      AvlNode* left_leaf_ptr = t->right;
      while (left_leaf_ptr -> left != nullptr)
        left_leaf_ptr = left_leaf_ptr -> left;
      //after the above loop, left_leaf_ptr is pointing to the left most
       ↪  leaf of t->right, attach t->left to the left subtree of this leaf
      left_leaf_ptr -> left = t->left;
    }

    //keep record of the address of current t->right
    AvlNode* temp = t->right;
    //reclaim memory
    delete t;
    //re-connect tree node
    t = temp;

  }

  //t is not pointing to the target node
  else if (t->element > val)
    remove(val, t->left);
  else if (t->element < val)
    remove(val, t->right);

  //balance the node to correct height-change induced un-balancing
   ↪  situation
  balance(t);
```

```
}
```

6.2.3.19 findMin()

```
template <typename comparable>
typename AVLTree<comparable>::AvlNode*
  ↪  AVLTree<comparable>::findMin(AvlNode* t) const {
  if (t == nullptr)
    return t;

  while (t->left != nullptr)
    t = t->left;
  //after the above loop, t is not pointing to left-most leaf
  return t;
}
```

6.2.3.20 findMax()

```
template <typename comparable>
typename AVLTree<comparable>::AvlNode*
  ↪  AVLTree<comparable>::findMax(AvlNode* t) const {
  if (t == nullptr)
    return t;

  while (t->right != nullptr)
    t = t->right;
  //after the above loop, t is not pointing to right-most leaf
  return t;
}
```

6.2.3.21 contains()

```
template <typename comparable>
bool AVLTree<comparable>::contains(const comparable& val, AvlNode* t) const
  ↪  {
  //base case: t == nullptr, no match found
  if (t == nullptr)
    return false;

  //base case2: t->element == val
  if (t->element == val)
    return true;

  //try to find val in t's children
  if (t->element > val)
    return contains(val, t->left);
```

```
14    else
15      return contains(val, t->right);
16  }
```

6.2.3.22 makeEmpty()

```
1   template <typename comparable>
2   void AVLTree<comparable>::makeEmpty(AvlNode* & t) {
3     //base case
4     if (t == nullptr)
5       return;
6
7     //begin makeEmpty
8     makeEmpty(t->left);
9     makeEmpty(t->right);
10    delete t;
11    t = nullptr;
12  }
```

6.2.3.23 printTree()

```
1   template <typename comparable>
2   void AVLTree<comparable>::printTree(AvlNode* t, std::ostream& out) const {
3     //base case
4     if (t == nullptr)
5       return;
6
7     //print the tree in inorder traversal
8     printTree(t->left, out);
9     out << t->element << ' ';
10    printTree(t->right, out);
11  }
```

6.2.3.24 clone()

```
1   template <typename comparable>
2   typename AVLTree<comparable>::AvlNode* AVLTree<comparable>::clone(AvlNode*
    ↪  t) const {
3     /** pay attention that what you clone is an AvlNode! **/
4     //base case
5     if (t == nullptr)
6       return t;
7
8     //clone
9     return new AvlNode{t->element, clone(t->left), clone(t->right)};
10  }
```

6.2.3.25 rotateWithLeftChild()

This function accepts a pointer to `AvlNode` type: `t`, and the AVL property of `*t` is violated. It will fix it by performing rotation of `t` with `t->left`. The process is illustrated in Figure 6.4. Pay attention that, besides rotating, you have to also update the height of each altered node, i.e. `t` and `t->left`. Then you reconnect `t->left` back to tree. `std::max()` is used to return the larger height of a node. And the node's height is obtained by calling `height()` routine. Code:

```
template <typename comparable>
void AVLTree<comparable>::rotateWithLeftChild(AvlNode* &t) {
  // rotate
  auto temp = t->left;
  t->left = temp->right;
  temp->right = t;

  // update height
  t->height = std::max(height(t->left), height(t->right)) + 1;
  temp->height = std::max(height(temp->left), t->height) + 1;

  // reconnect temp to where t was
  t = temp;
}
```

6.2.3.26 rotateWithRightChild()

This is similar with `rotateWithLeftChild()`. Code:

```
template <typename comparable>
void AVLTree<comparable>::rotateWithRightChild(AvlNode* &t) {
  AvlNode* temp = t->right;
  t->right = temp->left;
  temp->left = t;

  //update height information
  t->height = std::max(height(t->left), height(t->right)) + 1;
  temp->height = std::max(height(temp->right), t->height) + 1;

  //connect temp to where t was
  t = temp;
}
```

6.2.3.27 doubleWithLeftChild()

This function will perform a double rotation. The process is illustrated in 6.5. Notice that you can call single rotation routine to rotate (height change is also taken care of by single rotation routine). Code:

```
template <typename comparable>
void AVLTree<comparable>::doubleWithLeftChild(AvlNode* &t) {
  rotateWithRightChild(t->left);
  rotateWithLeftChild(t);
}
```

6.2.3.28 doubleWithRightChild()

This function will perform a double rotation. The process is illustrated in 6.6. Notice that you can call single rotation routine to rotate (height change is also taken care of by single rotation routine). Code:

```
template <typename comparable>
void AVLTree<comparable>::doubleWithRightChild(AvlNode* &t) {
  rotateWithLeftChild(t->right);
  rotateWithRightChild(t);
}
```

6.2.3.29 balance()

This function accepts a pointer to `AvlNode` type: `t`. It will first check if the AVL property is violated, if so, it will try to fix it using the four rotation routines. The algorithm is based on previous analysis. Pay attention that you sould update the height of the current node.

```
template <typename comparable>
void AVLTree<comparable>::balance(AvlNode* &t) {
  //check nullptr
  if (t == nullptr)
    return;

  //check which branch inserted
  if (height(t->left) > height(t->right) + ALLOWED_IMBALANCE) {// left
  ↪  subtree of t is higher
    if (height(t->left->left) < height(t->left->right)) //LR case
      doubleWithLeftChild(t);
    else //LL case
      rotateWithLeftChild(t);
  }

  else if (height(t->right) > height(t->left) + ALLOWED_IMBALANCE) {//right
  ↪  subtree of t is higher
    if (height(t->right->right) < height(t->right->left)) //RL case
      doubleWithRightChild(t);
    else //RR case
      rotateWithRightChild(t);
  }
```

```
21
22    //update height of the current node
23    t->height = std::max(height(t->left), height(t->right)) + 1;
24  }
```

6.2.3.30 height()

By default, the height of an empty AVL tree is -1. Code:

```
1  template <typename comparable>
2  int AVLTree<comparable>::height(AvlNode* t) const {
3    return t == nullptr ? -1 : t->height;
4  }
```

6.3 Red Black Tree

6.3.1 General Idea

Chapter 7

Hash Table

Hash table ADT is a way of organizing data. It only allows a subset of operations of binary search tree. However, Hash table has higher efficiency on these supported operations than binary search tree. For example, Hash table can perform insertion, deletion and find in **CONSTANT** average time.

One point should be made clear that, Hash table does not support operation that requires ordering information.

7.1 General Idea

7.1.1 Key

A key is part of the data item that is used to perform searching by some methods. We search the **key** and find the match item, then we declare that we have found the item.

7.1.2 Vector/Array

The ideal Hash table data structure is merely an array (which has a fixed size) containing our data items. Imagine we have a bunch of data items to be put into an array. In order to achieve constant time access, we may think design a certain rule of how to put items into the array. This rule can be a mapping from a specific key to the actual index in that array. We can define some part of our data item as the **KEY** to be used to get the index through the mapping. So, for each data item, we can find out where we should put it in the array. And we can also find out the existence of a specific data item by using the corresponding key in constant time.

Basically, hash table is an array that manages the position of data items by their key, rather than their sequence of inserting into the array. Figure 7.1 shows the comparison between array storage and hash table storage.

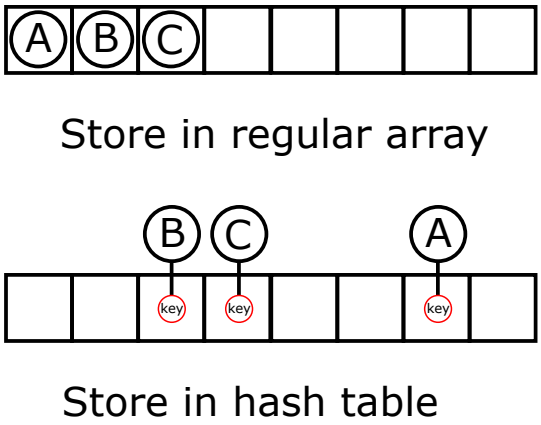

Figure 7.1: Comparison between array storage and hash table storage

7.2 Hash Function

The mapping we mentioned above is called a hash function. Ideally, a hash function should:

1. be simple to compute
2. can ensure that any two distinct keys get different index value

The fact that there are a finite number of cells and a virtually inexhaustible supply of keys makes 2 impossible. In practical, we try our best to achieve it. By that, it means seek a hash function that distributes the keys as even as possible among the slots of the array.

If the input keys are integers already, simpliy choose `hash(key)` as `key % array.size()`. If `array.size()` is a prime number, this hash function can distribute the keys evenly.

If the input keys are not integer, our strategy contains two steps:

1. convert the key into integer `i`
2. calculate `i % array.size()` to get the index

In practice, the function that fullfills step 1 is called hash function. The second step is trival.

7.2.1 Hash Function Object

We have to design the details of how hash function convert a non-integer data type into integer value. For example, to convert a string object to integer value. We can just directly write a function that accepts `std::string` and returns `size_t`:

```
size_t hash(const std::string& key) {
  size_t hashVal = 0;
  for (char ch:key)
    hashVal = 37 * hashVal + ch;
  return hashVal;
}
```

To do this job in a more generic way, we can define a hash function object. A function object (also called a functor) is a class object with `operator()` defined. This operator is called function call operator, which can make the object be used like a function. For example:

```
class Add {
public:
  int operator()(int a, int b) {
    return (a + b);
  }
};
```

In the above code, we defined a class named `Add`. It has only one public member function `operator()`, which accepts two integer parameters `a` and `b`. It will return the result of `a + b`. To use it, we can:

```
Add addition; // declare an object
std::cout << "1 + 1 is: " << addition(1, 1); // operator() is called
```

We can also write a template:

```
template <class T>
class Add {
public:
  T operator()(T a, T b) {
    return (a + b);
  }
};
```

To use it:

```
Add addition<int>; // declare an object
std::cout << "1 + 1 is: " << addition(1, 1); // operator() is called
```

You can also write a **specialization** of the template for a spefic type. For example:

```
// for generic type T
template <class T>
class Add {
public:
  T operator()(T a, T b) {
    return (a + b);
  }
};

// for string type
template<> // you'll provide the specific type below
class Add<std::string> { // provide type here
public:
  int operator()(std::string a, std::string b) {
```

```
15      return (std::stoi(a) + std::stoi(b));
16    }
17  };
```

This is called template specialization. You write the implementation of the class using one specific type. Pay attention that, if all the template typename is designated, it will be implemented during compile.

Back to our hash function object. We can first declare an empty template and write template specialization for different types of key we want to convert to integer. For example, in the following code, hash function for `std::string` and `double` are defined.

```
1   #pragma once
2   #include <string>
3   //an empty function object template
4   template <typename Key>
5   class hash {
6   public:
7     size_t operator()(const Key& k) const;
8   };
9
10  //a specialization of hash function, which accepts a string and return the
    ↪  converted integer value
11  template<>
12  class hash<std::string> {
13  public:
14    size_t operator()(const std::string& k) const {
15      size_t hashVal = 0;
16      for(char ch:k)
17        hashVal = 37 * hashVal + ch;
18      return hashVal;
19    }
20  };
21
22  //a specialization of hash function, which accepts a double and return the
    ↪  converted hash value (this is only for test purpose, so the hash
    ↪  function is bad)
23  template<>
24  class hash<double> {
25  public:
26    size_t operator()(double k) const {
27      if (double < 0)
28        double = 0 - double;
29
30      size_t hashVal{static_cast<size_t>(k)}; //requires narrowing conversion
31
```

```
32      return hashVal;
33    }
34  };
```

The above code is wrapped in a header file (this header file is considered to store the hash function). When using hash table, we may want to use it to hold user-defined data type, like an object of user-defined class. It is your duty to provide usable hash function specialization that can convert the user-defined class into integer value. Generally, this specialization of hash function should be inside the declaration file of the class that is going to be put into hash table (so as long as you included the class, you can use it in hash table container). The keyword of the name of hash function is just that: `hash()`.

For example, suppose I have a class named `Employee`. It has following private data member:

```
std::string name;
double salary;
```

Now I want to use `name` as the key to generate the hash-value of an employee object. So I have to put following template specialization of hash function into the header file of `Employee` class (employee.h):

```
1  template<>
2  class hash<Employee> {
3  public:
4    size_t operator()(const Employee& item) {
5      static hash<std::string> hf;
6      return hf(item.getName()); //this line makes it clear that, the
       ↪  returned value of item.getName() will be used as key of item, which
       ↪  is a string
7    }
8  };
```

Notice that, inside the implementation, I created an object of `hash<std::string>` type. This is to utilize the hash function that accepts a `std::string` type, an example of using hash function for pre-defined types to build our own hash function.

The keyword `static` is just tell computer that only one copy of the `hash<std::string>` object is needed in case of multiple calling of `hash<Employee>::operator()`. `getName()` is the routine in `Employee` class that returns `name`.

7.2.2 Hash Function in C++ STL

Designing a good hash function is usually very hard in practical, and is usually done by mathematicians working in related field. In C++ STL, we have predefined hash function for primitive types as well as types predefined in C++ STL (like `std::string`). To use it, you have to:

```
#include <functional>
```

Hash function for `std::string` object is defined in `<string>` header file. Pay attention that you still need to write the specific version of hash function that accepts your own user-defined class (by giving a template specialization in the user-defined class).

Using the same example in the previous section, the following code shows how to implement the hash function specialization for `Employee` class using C++ STL provided hash function instead:

```
#include <string> // enable C++ STL hash function for string
template<>
class hash<Employee> {
public:
  size_t operator()(const Employee& item) {
    static std::hash<std::string> hf;
    return hf(item.getName()); //this line makes it clear that, the
    ↪  returned value of item.getName() will be used as key of item, which
    ↪  is a string
  }
};
```

7.3 Collision Management

Collision happens when inserting an element, it hashes to the same value as a previously inserted element. The probability of collision increases as the hash table gets full. There are various ways to manage collision, in this section following will be introduced briefly:

- separate chaining
- probing
  - linear probing
  - quadratic probing
- double hashing

7.3.1 Separate chaining

This strategy will keep a doubly-linked list (`std::list`) of all elements that hash to the same value. The hash table array will not store data element directly, it keeps track of a doubly linked list of data element in the array instead. If collision happens, data element hashes to the same index value will be pushed into the list. Actually, any containers can be used beside linked list, for example: binary tree, another hash table.

Load factor λ is defined as:

$$\lambda = \frac{\text{total number of items in hash table}}{\text{hash table size}} = \frac{\text{total number of items}}{\text{number of linked list}}$$

so, load factor is also the average length of the linked list. A successful search requires about $1 + \frac{\lambda}{2}$ links to be traversed. $\frac{\lambda}{2}$ corresponds to expected number of other elements traversed before we find match, 1 corresponds to the matched element.

To reduce the chance of collision, we want to keep λ low. When $\lambda \approx 1$, we may want to resize the hash table. When copying old items into new array, we have to use hash function to re-calculate the appropriate index for each of the item. This process is called rehash.

7.3.2 Probing

In this strategy, if an element is hased to a slot that is already occupied, we try alternative slots until an empty cell is found. The operation of trying to find an empty alternative slot is called probing. So, how to calculate these "alternative" index value if the first hashed value is already occupied?

We list the alternative indexes as:

$$h_0(x), h_1(x), h_2(x), \cdots$$

where

$$h_i(x) = [hash(x) + f(i)] \text{ mod tableSize}$$

with $f(0) = 0$.

The function $f(i)$ is called the collision resolution strategy. A hash table doesn't use separate chaining requires a larger table than those use. Generally, λ should be kept below 0.5.

7.3.2.1 Linear probing

Linear probing is to use linear function of i, typically $f(i) = i$. Basically, if one spot is occupied, we move to the adjacent spot. Repeat this process until we find an empty spot. As long as the table is big enough, an empty spot can clways be found. However, the time to do so cen get quite large (for both insertion and searching). Another issue is blocks of occupied slots start forming quickly, which is known as primary clustering.

7.3.2.2 Quadritic probing

Quadritic probing is to use quadratic function of i, typically $f(i) = i^2$. If the table size is prime number, there is no guarantee of finding an empty cell once the table gets more than half full. This is because at most half of the table can be used as alternative location to resolve collisions for objects that hash to same index value in the begining ($h_o(x)$). Formally, we have the following theorem:

THEOREM

> If quadratic probing is used, and the table size is prime, then a new element can **always** be inserted if the table is at most half empty

PROOF

Intuitively, we can imagine why we are facing this problem: even if the table is not full, we still can't insert an item. Unlike linear probing, quadratic probing is not probing empty slot **one by one**. For linear probing, as long as there is empty cell, we can always get there. For quadratic probing, we are skipping some empty slots. In fact, when inserting an element, we first calculate $h_0(x)$, then all the possible $h_i(x)$ can be calculated:

$$\begin{aligned} h_0(x) &= hash(x) \bmod \text{tableSize} \\ h_1(x) &= (hash(x) + 1) \bmod \text{tableSize} \\ h_2(x) &= (hash(x) + 4) \bmod \text{tableSize} \\ &\vdots \\ h_i(x) &= (hash(x) + i^2) \bmod \text{tableSize} \end{aligned}$$

the fact is, the sequence $h_0(x), h_1(x), h_2(x), \cdots, h_i(x), \cdots$ has finite number of distinct terms. The number of distinct terms is the number of items you can insert into the hash table.

To demonstrate the occurence of duplication of $h_i(x)$, assume the size of the table is a prime number N. Let $i = 0, 1, 2, \cdots$. We calculate $h_i(x)$ all the way from $i = 0$ to $i = \lfloor \frac{N}{2} \rfloor$, since N is prime number, this is equal to $\frac{N-1}{2}$. We have:

$$h_i(x) = [hash(x) + (\frac{N-1}{2})^2] \mod N$$

Then, we calculate $h_{i+1}(x)$:

$$h_{i+1}(x) = [hash(x) + (\frac{N+1}{2})^2] \mod N$$

then, $h_i(x) - h_{i+1}(x)$ is:

$$\begin{aligned} &= (\frac{N-1}{2})^2 \mod N - (\frac{N+1}{2})^2 \mod N \\ &= (N^2 - 2N + 1) \mod N - (N^2 + 2N + 1) \mod N \\ &= 1 \mod N - 1 \mod N \\ &= 0 \end{aligned}$$

So, $h_i(x) = h_{i+1}(x)$, duplicate occured.

Similarly, $h_{i+2}(x) = h_{i-2}(x)$, if $i = \lfloor \frac{N}{2} \rfloor$. Thus, $i = 0, 1, 2, \cdots, \lfloor \frac{N}{2} \rfloor$ is distinct alternative locations, total number is:

$$\lfloor \frac{N}{2} \rfloor + 1 = \lceil \frac{N}{2} \rceil$$

For an arbitrarily $hash(x)$ value, the first $\lceil \frac{N}{2} \rceil$ alternative locations are distinct. If a hash table has less than $\lceil \frac{N}{2} \rceil$ positions occupied, an empty spot can always be found for any item inserted.

7.3.3 Double hashing

In this strategy, when an collision occurs, we use another hash function to probe:

$$h_i(x) = [hash(x) + f(i)] \text{ mod tableSize}$$

and:

$$f(i) = i \cdot hash_2(x)$$

The function $f(i)$ should never evaluate to zero, otherwise, $h_i(x) = hash(x)$ mod tableSize will occur. A typical choice of $hash_2(x)$ is:

$$hash_2(x) = R - (x \bmod R)$$

where R is a prime number smaller than tableSize.

7.4 Simple Implementation

7.4.1 Separate Chaining

We'll implement a hash table class template in this section. It uses separate chaining strategy to manage collisions. It requires the object stored in it can provide a specialization of `hash()` function, so it can use it to generate index value. Take employee class as an example.

```
#pragma once
#include <iostream>
#include <string> // for hash<std::string>

class Employee {
private:
  std::string name; //name will be used as key
  double salary;

public:
  //constructor
  Employee(std::string n = "N/A", double s = 0) : name{n}, salary{s} {}

  //name accessor
  const std::string& getName() const {
    return name;
  }

  //salary accessor
  double getSalary() const {
    return salary;
  }
```

```
23
24    //equality operator
25    bool operator==(const Employee& rhs) const {
26      return getName() == rhs.getName();
27    }
28
29    //non-equal operator
30    bool operator!=(const Employee& rhs) const {
31      return !(*this == rhs);
32    }
33
34    //overload << to enable printing class info
35    friend std::ostream& operator<<(std::ostream& oi, const Employee& obj) {
36      oi << "Name: " << obj.name << "   Salary: $ " << obj.salary << '\n';
37      return oi;
38    }
39  };
40
41  //define a version of hash function that specifically accepts this Employee
    ↪  type, and return the "integerized" key of this Employee class.
42  //this implementation is in the Employee class declaration file, which
    ↪  means the Employee class objects provide a hash function that
    ↪  specifically work for them
43  template<>
44  class hash<Employee> {
45  public:
46    size_t operator()(const Employee& item) {
47      static std::hash<std::string> hf; // declare an hash function object
48      return hf(item.getName()); //the returned value of item.getName() will
        ↪  be used as key of item, which is a string
49    }
50  };
```

Notice that we have provided the `hash()` function specialization in the `Employee` class, so we can use it in our hash table class.

The header for our hash table class is as follows:

```
1  #pragma once
2  #include <vector>
3  #include <list>
4  #include <algorithm> //for std::find()
5  #include <functional> //for std::hash() function
6
7  template <typename HashedObj>
8  class HashTable {
```

```
public:
  //accepts an integer value, will build up a vector of that size, and
  ↪  initialize each entry an empty list (by calling the default
  ↪  constructor of list)
  explicit HashTable(int size = 101);

  bool contains(const HashedObj& x) const;

  void makeEmpty();
  bool insert(const HashedObj& x);
  bool insert(HashedObj&& x);
  bool remove(const HashedObj& x);

private:
  std::vector<std::list<HashedObj>> theLists; //array of lists
  int currentSize; // hold current number of items in array

  //void rehash();
  size_t myhash(const HashedObj& x) const;
};

#include "ht_separate_chain.hpp"
```

Notice that we have declared a vector of list as our container for the hash table. The implementation is given below.

7.4.1.1 Constructor

It accepts an integer value, will build up a vector of that size, and initialize each entry an empty list (by calling the default constructor of list).

```
template <typename HashedObj>
HashTable<HashedObj>::HashTable(int size) {
  for (int i = 0; i < size; ++i)
    theLists.push_back(std::list<HashedObj>{});

  currentSize = 0; //initialize the size, or do we have to specifically
  ↪  code it? should be 0 automatically
}
```

7.4.1.2 contains()

this function tries to find if there is an element `x` stored in hash table. We do it in following ways:

- call `myhash(x)` to find out the index of `x`

- navigate the vector of lists to locate the list that should contain `x`, if `x` was inserted
- use `std::find()` to check if `x` exist in that list

```
template <typename HashedObj>
bool HashTable<HashedObj>::contains(const HashedObj& x) const {
  auto& whichList = theLists[myhash(x)]; // whichList is reference to list!
  return std::find(whichList.begin(), whichList.end(), x) !=
   ↪  whichList.end();
}
```

7.4.1.3 `makeEmpty()`

This function will clear all lists stored in the vector of lists, and also reset the `currentSize` to 0.

- use a ranged for loop to traverse each list stored in theLists.
- use `std::list::clear()` to clear each list
- reset `currentSize`

```
template <typename HashedObj>
void HashTable<HashedObj>::makeEmpty() {
  for (auto& thisList : theLists)
    thisList.clear();

  currentSize = 0; // reset value of currentSize
}
```

7.4.1.4 `insert()`

This function will insert an element `x` into the list. Steps are as follows:

- call `myhash(x)` to find out the index of `x`
- navigate the vector of lists to locate the list that should contain `x`, if `x` was inserted
- use `std::find()` to check if `x` exist in that list
- if `x` is not in the list
  - call `push_back(x)` to insert it into the list
  - update `currentSize`
  - check the load factor of hash table
    - load factor $>$ 1: call `rehash()`
    - otherwise, do nothing
- if `x` is already in the list, return `false` to indicate insertion failed.

The code is as follows (copy version):

```
template <typename HashedObj>
bool HashTable<HashedObj>::insert(const HashedObj& x) {
  auto& whichList = theLists[myhash(x)];

  auto itr = find(whichList.begin(), whichList.end(), x);

  if (itr == whichList.end()) {
    whichList.push_back(x);

    if (++currentSize > theLists.size())//update size and check if rehash
    ↪  is needed
      rehash();

    return true;
  }

  return false; //match found
}
```

7.4.1.5 `remove()`

This function will remove an element `x` in the list. Steps are as follows:

- call `myhash(x)` to find out the index of `x`
- navigate the vector of lists to locate the list that should contain `x`, if `x` was inserted
- use `std::find()` to check if `x` exist in that list
- if `x` is not in the list
  - return `false` to indicate erase failed
- if `x` is found in the list
  - call `std::list::erase()` to erase `x`
  - update `currentSize`
  - return true to indicate erase succeed

Code:

```
template <typename HashedObj>
bool HashTable<HashedObj>::remove(const HashedObj& x) {
  auto& whichList = theLists[myhash(x)];

  auto itr = std::find(whichList.begin(), whichList.end(), x);

```

```
  if (itr == whichList.end())
    return false;
  else {
    whichList.erase(itr);
    --currentSize;
    return true;
  }
}
```

7.4.1.6 myhash()

This function accepts an element `x`, and will return the hashed index value for the input `x`. It requires a proper specialization of `hash()` defined for `x`. Code:

```
template <typename HashedObj>
size_t HashTable<HashedObj>::myhash(const HashedObj& x) const {
  static hash<HashedObj> hf;
  return hf(x) % theLists.size();
}
```

7.4.1.7 rehash()

When the load factor ≈ 1, we should increase the hash table size to avoid performance reduction of the hash table (i.e. multiple collisions happen). The steps are as follows:

- declare a temporary vector of list (`temp`) to hold the current array (copy)
- call `resize(2 * array.size())` to expand the current array's size
- call `makeEmpty()` to clear up the current array
- use a ranged for loop to traverse each element in `temp`, call `insert()` to insert them into the new array

Code:

```
template <typename HashedObj>
void HashTable<HashedObj>::rehash() {
  // create copy of old lists
  std::vector<std::list<HashedObj>> temp = theLists;

  // expand the old list and make empty
  theLists.resize(2 * theLists.size());
  makeEmpty();

  // copy back
  for (auto& list : temp) // for each list in the old theLists
    for (auto& x : list) // for each element in list
```

```
13        insert(std::move(x)); // insert back, use move version
14    }
```

7.4.2 Quadratic Probing

In this section, a hash table with quadratic probing collision management strategy will be implemented. Similar with the separate chaining example, we'll implement this hash table to hold generic type `HashedObj`, which is required to provide the specialization of `hash()` function.

The public interface is:

```
explicit HashTable(int size = 101);
bool contains(const HashedObj& x) const;
void makeEmpty();
bool insert(const HashedObj& x);
bool insert(HashedObj&& x);
bool remove(const HashedObj& x);
enum EntryType {ACTIVE, EMPTY, DELETED};
```

Notice that, an enumerated data type named `EntryType` has been declared (in header file of this hash table class). This will be used to indicate the status of a certain slot in hash table array. Why we need this? In probing strategy, we probe the slot to find available empty slot. We need the slot of the array hold not only `HashedObj`, but also information on this slot: is it empty? is it actively storing a `HashedObj`? or is it deleted (so we can insert new `HashedObj` to it). This demand prompts us to use an object to wrap our `HashedObj` and variable that can indicate the various states of where this object stored, and we insert this object into the slot of the hash table array. The detail of the object is shown below:

```
struct HashEntry {
  HashedObj element; //hold the hashed object
  EntryType info; //hold the current status of the item

  HashEntry(const HashedObj& e = HashedObj{}, EntryType i = EMPTY) :
  ↪  element{e}, info{i} {}

  HashEntry(HashedObj&& e, EntryType i = EMPTY) : element{std::move(e)},
  ↪  info{i} {}
};
```

This declaration of structure named `HashEntry` is in private section of the hash table class. Notice that we have declared a variable named `info`, whose type is the enumerated type we have just defined: `EntryType`. The object of this struct will be stored in the entry of hash table array.

Now let's take a look at the private member of our hash table class:

```
// member
```

```
std::vector<HashEntry> array;
int currentSize;
// member functions
bool isActive(int currentPos) const;
int findPos(const HashedObj& x) const;
void rehash();
size_t myhash(const HashedObj& x) const;
```

Notice that the type of the hash table array is a vector of `HashEntry` type. We also have a `currentSize` member to hold the number of `HashedObj` in the hash table. Previous analyze indicates that we must keep the loading factor less than 0.5 to ensure successful inserting for a new element.

Let's take a look at the implementation.

7.4.2.1 `constructor`

The constructor accepts an integer value, which will be used to declare the underlying vector. Also, `currentSize` should be initialized to zero. Code:

```
template <typename HashedObj>
HashTable<HashedObj>::HashTable(int size) : array(size) {
  currentSize = 0;
}
```

7.4.2.2 `contains()`

This function accepts a `HashedObj` type parameter `x`, its working steps are as follows:

- call `findPos(x)` to get the index of `x`
- call `isActive()` to check if `array[index]` is active

The index value returned by `findPos(x)` is the **SHOULD** index of `x` in the current array. If `x` is in the array, this index should be its index, however the status may not be `ACTIVE` (so we need to call `isActive` to determine). If `x` is not in the array, this index should be where it is inserted (if you are inserting `x`), so the status should be either `DELETED` or `EMPTY`. By checking the status of `array[index]`, we know whether `x` is in the hash table or not. Code:

```
template <typename HashedObj>
bool HashTable<HashedObj>::contains(const HashedObj& x) const {
  return isActive(findPos(x));
}
```

7.4.2.3 `makeEmpty()`

This function will empty the entire hash table array. We use lazy deletion to achieve this task (we are using enumerated type to label the status of each slot, so changing the status

to do lazy deletion is very intuitive). All we have to do is to reset `currentSize` and modify the status label of each slot to `EMPTY`. Code:

```
template <typename HashedObj>
void HashTable<HashedObj>::makeEmpty() {
  currentSize = 0;
  for(auto& entry : array) //range based for loop
    entry.info = EMPTY;
}
```

7.4.2.4 `insert()`

This function accepts a `HashedObj` type parameter `x`. It will insert `x` into the hash table. Working steps:

- call `findPos(x)` to find out the **SHOULD** index of `x`
- check if `x` is already in the hash table
  - yes: return false (insertion failed: already in)
- copy `x` to the slot, and set status as `ACTIVE`
- rehash if load factor is greater than 0.5
- return true to indicate insertion succeed

Code (copy version):

```
template <typename HashedObj>
bool HashTable<HashedObj>::insert(const HashedObj& x) {
  int currentPos = findPos(x);
  if (isActive(currentPos))
    return false; //already in the table

  array[currentPos].element = x;
  array[currentPos].info = ACTIVE;

  //rehase
  if (++currentSize > array.size() / 2)
    rehash();

  return true; //insertion successful
}
```

7.4.2.5 `remove()`

This function search and removes the entry that is equal to passed in parameter `x`. Working steps:

- call `findPos(x)` to find out the **SHOULD** index of `x`
- check if `x` is in the hash table
  - no: return false (remove failed: `x` not found)
- lazy deletion: set status as `DELETED`
- return true to indicate remove succeed

Code:

```
template <typename HashedObj>
bool HashTable<HashedObj>::remove(const HashedObj& x) {
  int currentPos = findPos(x);
  if (!isActive(currentPos))
    return false; //current position hold no active object (either deleted
    ↪  or empty)

  array[currentPos].info = DELETED;
  return true;
}
```

7.4.2.6 `isActive()`

This function accepts an integer type `currentPos`, which is the index to check. If the status is `ACTIVE`, return true, otherwise, return false. Code:

```
template <typename HashedObj>
bool HashTable<HashedObj>::isActive(int currentPos) const {
  return array[currentPos].info == ACTIVE;
}
```

7.4.2.7 `findPos()`

This is one of the most important function in our hash table class. It accepts a `HashedObj` type parameter `x`, and will return the **SHOULD** index of `x`. As mentioned above, this index is where `x` ashould be in the current hash table. If `x` is currently in the hash table, then this index is where it stored (the status may be `ACTIVE` or `DELETED`, though). If `x` is not in the hash table, then this index is where it should be inserted (i.e. this is the first non-ACTIVE slot of the series of available slots for `x`).

The working steps are as follows:

- pass `x` to myhash() function, the myhash() function will prompt the call of the HashedObj's own version of hash function, to convert a HashedObj to an size_t type integer based on the logic defined in the HashedObj class. Pay attention that this integerized value is raw — doesn't mod the size of array yet (doesn't scaled). This is because in the final index looking iteration, we have to add i^2 to this value, and then mod the array size ($h_i(x) = (hash(x) + i^2)$ mod tableSize).

We store `myhash(x) % array.size()` to `currentPos`.

- use a variable `stepSize` to hold the current number of hash value finding iteration. This is to calculate the next index value using the quadratic rule.
- use a while loop to check whether: 1. slot at `currentPos` is labeled `EMPTY`; 2. slot at `currentPos` is storing `x`. These are two stopping conditions for the while loop, if one of them is true, then `currentPos` is the **SHOULD** position of `x`.
- if `currentPos` is not where `x` should be, we update it:
  - calculate the next **SHOULD** position
  - update `stepSize`
- after the while loop, we return the **SHOULD** index. Notice that we will always be able to find one, because we always `rehash()` if the loading factor is greater than 0.5.

Code:

```
template <typename HashedObj>
int HashTable<HashedObj>::findPos(const HashedObj& x) const {
  int stepSize = 0;
  int initialPos = myhash(x);
  int currentPos = (initialPos + stepSize * stepSize) % array.size();

  while (array[currentPos].info != EMPTY && array[currentPos].element != x)
  ↪ {
    stepSize++;
    currentPos = (initialPos + stepSize * stepSize) % array.size();
  }

  return currentPos;
}
```

7.4.2.8 `rehash()`

Similar with `rehash()` in separate chaining structure, we need to keep the old array first, then we expand the size of the current array. Then we insert back elements into the new array. One thing should be made clear is that, we only have to insert those `ACTIVE` elements back to the new array.

Working steps:

- declare `temp`, a vector of `HashEntry` type, and copy old array to it
- expand array and make empty
- use a range based for loop to traverse `temp`, insert those elements that are `ACTIVE`

Code:

```
template <typename HashedObj>
void HashTable<HashedObj>::rehash() {
  auto temp = array;

  array.resize(array.size() * 2);
  makeEmpty();

  for (auto& x : temp)
    if (isActive(x))
      insert(std::move(x));
}
```

Chapter 8

Priority Queue

8.1 General Idea

A priority queue is a special kind of queue. Elements in priority queue is weighted—they have an attribute to be used to determine the sequence they leave the queue. In this section, we assume the smallest element leaves the queue first.

A priority queue is a data structure that allows at least the following two operations:

- `insert`
- `deleteMin`

`insert` does the obvious thing: insert an item into the data structure (container). `deleteMin` will finds, returns and removes the minimum element in the priority queue.

There are various ways to implement a priority queue. For example, we can implement a binary search tree. So whenever we `deleteMin`, we can just delete the left most element in the binary tree (because it is the smallest element). Whenever we need to `insert`, we just call the `insert()` routine to do so.

However, this is not the ideal way because we may add complexity to the problem. It seems like some kind of overkill: we not only satisfy the requirement of prority queue, we also end up with a totally ordered data structure. In certain circumstances, we really only need the minimum. Keeping all other information seems like a waste of resources.

8.2 Simple Implementation

Let's see a simple implementation of priority queue. We'll use a tree structure to implement the priority queue. Since binary search tree is overkill, we can implement a partially ordered binary search tree to achieve our goal, this is also called a binary **heap**.

8.2.1 Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right, such a tree is known as a **complete binary tree**.

It can be shown that a complete binary tree can be represented by an array (without the need of link). This is by knowing a node's position in the array (the index), we can calculate the position of its parent, left child and right child. Specifically:

- if the root node's index is 1, then for a node at `array[i]`:
  - its parent is at `array[i/2]`
  - its left child is at `array[2*i]`
  - its right child is at `array[2*i+1]`
- if the root node's index is 0, then for a node at `array[i]`:
  - its parent is at `array[(i+1)/2-1]`. This does not apply to root node.
  - its left child is at `array[2*i+1]`
  - its right child is at `array[2*i+2]`

Now, we define the **partial order** of the complete binary tree. This **partial order** allows operations of priority queue to be performed quickly (its also called **heap-order-property**). Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node should be smaller than **ALL** of its descendants. Apply this logic, we arrive at the heap-order property: in a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X, with the exception of root. This property suggests the minimum element can always be found at the root. Thus, `findMin` can operate in constant time.

8.2.2 Binary Heap Class

Based on the above discussion, we can use an array to keep our heap. `std::vector` is a good choice. In our heap class, we also want to use an integer to keep track of the number of elements in heap: `currentSize`. Also, we'll use `array[1]` as the slot to hold the root of the heap (so `array[currentSize]` is the last element in the heap).

8.2.3 `insert()`

Pay attention to following things:

- before adding a new element to a heap, always check the capacity of current container (the vector).
- new item can only be added to the rightmost slot at the bottom of the complete binary tree.

- after adding a new item, the heap-property may be violated. For example, the newly inserted item is smaller than its parent.
- if heap-property is violated, we percolate up until:
  - we find the proper slot for inserted item, or:
  - we reach the root

Code for `insert()` routine:

```
template <class T>
void BinaryHeap<T>::insert(const T& x) {
  if (array.size() == currentSize + 1) {
    array.resize(array.size() * 2);
  }

  int hole_index = ++currentSize;
  while (hole_index != 1 && x < array[hole_index/2]) {
    array[hole_index] = std::move(array[hole_index/2]);
    hole_index /= 2;
  }

  array[hole_index] = x;
}
```

8.2.4 deleteMin()

The advantage of a heap is it can access it's smallest element at constant time, because it is stored at the root position (or the first item in the underlying array). On the other hand, heap is a complete binary tree, which means any addition or deletion should occur at the leftmost slot of the bottom layer (i.e. the last element in the underlying array).

To delete the smallest item, we remove the element stored in hole and use `array[currentSize]` to fill into the hole. This operation will very likely violate the heap order, since preciously root holds the smallest element in the heap, now an element in bottom layer (which is larger than at least half of the heap) is placed into root. So, we have to re-build heap property.

To re-build heap property, we use a routine `percolateDown()`. It accepts the position of a hole, and try to move it down to where it fits (i.e. to where the element in that hole satisfies heap order). This routine will be introduced in the following section. The code for `deleteMin()` is as follows:

```
bool BinaryHeap<T>::deleteMin() {
  //check if the heap is empty
  if (isEmpty())
    return false;

  //move the last item to the root
```

```
7   array[1] = std::move(array[currentSize--]);
8
9   //percolate down
10  percolateDown(1);
11
12  return true;
13 }
```

8.2.5 percolateDown()

The header of `percolateDown()` is as follows:

```
template <typename T>
void BinaryHeap<T>::percolateDown(int hole)
```

It accepts an integer identifying the position of a hole (index of this hole in the underlying vector). This function will check if element stored here violates the heap order. If so, it will move it downward until the element fits in.

`percolateDown()` will compare the element stored in hole with its the smaller one of its children, until:

- a position is found so that element in hole is smaller than its child or children;
- the process of percolating down has reached to the bottom layer, the hole doesn't have child, this is where it should go

First, we define a child index:

```
int child;
```

Then, we keep the value of element stored in the hole:

```
T tmp = std::move(array[hole]);
```

We use a for loop to percolate down. We compare the value of `tmp` with the value of its smaller child. If `tmp` is larger, we move it downward one layer, and lift the corresponding child up one layer. It is like you move the hole downward one layer. If `tmp` is smaller, then the current hole is the right spot to place `tmp` in, so we break the loop and place `tmp` to this spot.

The code for this function is as follows:

```
1  template <typename T>
2  void BinaryHeap<T>::percolateDown(int hole) {
3    int child;
4
5    T tmp = std::move(array[hole]);
6
7    for (; hole * 2 <= currentSize; hole = child) {//pay attention you may
     ↪  not have right child!
```

```
8       child = hole * 2; // select left child
9
10      // check if right child exist && if it is smaller than left child
11      if (child != currentSize && array[child + 1] < array[child])
12        ++child;
13
14      // tmp is larger than its smaller child, should percolate down
15      if (array[child] < tmp)
16        array[hole] = std::move(array[child]);
17      else //hole position found
18        break;
19    }
20
21    // place tmp into the proper hole position
22    array[hole] = std::move(tmp);
23  }
```

8.2.6 buildHeap()

8.2.6.1 Analysis and Implementation

Imagine we have an input array of size N. The arrangment of elements in this array is random. Now we want to build up heap order in this array, i.e. we want to rearrange the elements in this array so that they have heap property — a complete binary tree that is represented by array, and each parent is smaller than its children (if it has any).

In this section, a method with $O(N)$ complexity will be introduced. First, we load the input array into our internal array. Assume the height of the heap is h. Start from $(h-1)$ layer (because node in h layer does not have any child), for each node in the heap that has at least one child, we try to percolate it down. We do this until the final node: the root. Then the whole array is built with heap order.

So, our next question is: what is the index for the first node that has at least one child (start from $h-1$ layer).

To calculate this, first imagine a complete binary heap tree with N elements and the height is h. In hth layer, only leaf presents, and they may not occupy the whole layer. The number of node in layer h is:

$$\text{total number of nodes} - \text{number of nodes before layer } h = N - 2^h + 1$$

This number is either even $(2k)$ or odd $(2k+1)$. If its even, it means all their parents in layer $h-1$ have two children. If its odd, it means the parent of the last node has only one child.

Thus, we can express the number of node in layer $(h-1)$ that has at least one child as:

$$\lceil \frac{N - 2^h + 1}{2} \rceil$$

On the other hand, from the definition of complete binary tree, all nodes at layer 0 to layer $h-2$ have two children, i.e. $2^{h-1}-1$ nodes have two children.

Thus, the number of nodes that has at least one child in a complete binary tree of size N is:

$$2^{h-1}-1+\lceil\frac{N-2^h+1}{2}\rceil = 2^{h-1}-2^{h-1}+\lceil\frac{N-1}{2}\rceil$$

If N is even, $\lceil\frac{N-1}{2}\rceil = \frac{N}{2}$. If N is odd, $\lceil\frac{N-1}{2}\rceil = \lfloor\frac{N}{2}\rfloor$, which is `N/2` in C++ integer division rule.

If we place the root in `array[1]`, then `N/2` is the index of the last node in heap that has at least one child, which needs percolateDown.

The implementation is simple:

```
template <typename T>
void BinaryHeap<T>::buildHeap() {
  for (int i = currentSize / 2; i > 0; --i)
    percolateDown(i);
}
```

8.2.6.2 Complexity Analysis

The time complexity of `buildHeap()` can be bound by calculating the following terms:

- The time required to do a single percolate down
- The total number of percolate down required for all nodes

For a single percolate down, it will compare the two child of one node to pick the smaller one. And then it will compare the smaller child with the node. These are constant time complexity.

For the total number of percolate down we need, we have to consider the worst case: we have to percolate down each node we encounter to bottom. To calculate the total number is the same to calculate the sum of the heights of all the nodes in the heap, and it can be bound by the sum of the heights of all the nodes in a full-occupied complete binary tree of the same height. For such a tree, we have the following relation:

layer	number of nodes	height of nodes on this layer
0	1	h
1	2	h - 1
2	2^2	h - 2
⋮	⋮	⋮
h - 1	2^{h-1}	1
h	2^h	0

Thus, sum of heights for all nodes are:

$$2^0 \cdot h + 2^1(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(h-(h-1)) + 2^h \cdot 0 = \sum_{i=0}^{h} 2^i(h-i)$$

Let $S = \sum_{i=0}^{h} 2^i(h-i)$, then $2S = \sum_{i=0}^{h} 2^{i+1}(h-i)$. So:

$$2S - S = -h + 2^1 + 2^2 + \cdots + 2^h = -h + \frac{2 - 2^{h+1}}{1-2} = 2^{h+1} - h - 2$$

Notice the relation between N and h: $N = 2^{h+1} - 1$, so the time complexity for `buildHeap()` is:

$$O(N)$$

8.2.6.3 Build Heap by `insert()`

Another way to build heap is to traverse the input array, for each element encountered, we call `insert()` routine to insert it into our heap. From the implementation of `insert()` we know that, each inserted element will be percolated up to its proper position. Intuitively, this method has a higher time complexity than the percolate down method we just introduced. Let's call this `insert()` method as B, the percolating down method as A. We know:

- For A, all the nodes at $h-1$ level only need to percolate down **ONE** layer
- For B, only the second inserted node can enjoy one-level-percolating operation. For every newly inserted node, it has to percolate up to the root (the worst case).

Actually, for A and B, the number of percolating operation (down for A, up for B) of a specific node at a certain layer is just the opposite. We have the following relation:

layer	# of nodes	# of percolating (A)	# of percolating (B)
0	1	h	0
1	2	h - 1	1
2	2^2	h - 2	2
⋮	⋮	⋮	⋮
h - 1	2^{h-1}	1	h - 1
h	2^h	0	h

Total number of percolating in A is:

$$1 \cdot h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(h-(h-1)) + 2^h(h-h) = 2^{h+1} - h - 2$$

Total number of percolating in B is:

$$1 \cdot 0 + 2 \cdot 1 + 2^2 \cdot 2 + \cdots + (h-1)2^{h-1} + h \cdot 2^h = (h-1)2^{h+1} + 2$$

Now, plug in the relation between h and N:

$$h = \log N + 1 - 1$$

The number of percolating in A is: $N - \log N + 1 = O(N)$.

The number of percolating in B is: $(N + 1) \log N + 1 - 2N = O(N \log N)$.

Thus, building heap by calling `insert()` repeatedly has a much higher time complexity than the percolating down method.

Part III

Sorting

Chapter 9

Bubble Sort

9.1 General Idea

Bubble sort is a sorting algorithm that compares adjacent elements in the array. At each iteration of bubble sort, we compare the consecutive adjacent element. If they are out-of-order, we swap them to give order to the two adjacent elements, and label this iteration has **swapped**. Do this for all elements in the array. If one iteration has **swapped**, we do another. If we run through the array and no swapping happend, we know that the array is in order.

9.2 Implementation

The code is as follows:

```
1  template <class RandomAccessIterator, class Compare>
2  void Sort::bubbleSort(RandomAccessIterator first, RandomAccessIterator
   ↪  last, Compare comp) {
3    //define a variable to hold swapping information
4    bool swapped;
5
6    //go over the range and do bubble sort
7    do {
8      swapped = false; //at beginning of each iteration, label swapped as
       ↪  false
9      for (auto itr = first; itr < last - 1; ++itr) {
10       if (!comp(*itr, *(itr + 1)) && *itr != *(itr + 1)) {
11         std::swap(*itr, *(itr + 1));
12         swapped = true;
13       }
14     }
15
```

```
16    } while (swapped);
17  }
```

Pay attention that the function object `comp` may not define cases that two elements are equal. You have to manually consider this situation, otherwise, you may run into an infinite loop, since two elements equal will be considered **NOT** in order, and swap them won't change this fact, so the loop will not stop.

Chapter 10

Insertion Sort

10.1 General Idea

Given a random array with N elements, insertion sort will build a sorted sub-array at the front of the array by continuosly insert elements from the rest of the array into this ordered sub-array. Assume we're at pth iteration of insertion sort, then the subarray from `array[0]` to `array[p - 1]` are sorted, and we insert `array[p]` into the proper position in the ordered subarray, resulting an ordered subarray from `array[0]` to `array[p]`. We repeat this process until we traverse the array and have them sorted.

10.2 Implementation

Assume we are sorting an array in the range defined by two iterators: `[first, last)`. We'll take a look at pth iteration to determine how we insert. Assume we have an iterator `p` that is pointing to the element to be inserted. We use another iterator `i` to record its proper position (initially, `i = p`). We compare `*i` and `*(i - 1)` repeatedly (`i > first`). Two possible cases:

- `*(i - 1)` and `*i` are in order: we have found the proper position
- `*(i - 1)` and `*i` are not in order: we swap `*(i - 1)` and `*i` to give them order, and we update `i` (`--i`), then compare `*i` and `*(i - 1)` again.

After the above loop, we can accomplish the job of inserting `*p` into the proper position in the ordered subarray (by a bubble-sort like swapping).

We repeat this operation to all elements from element at `first + 1` to `last - 1`.

The code is as follows (a function object is used to define order):

```
1 template <class RandomAccessIterator, class Compare>
2 void Sort::insertionSort(RandomAccessIterator first, RandomAccessIterator
  ↪  last, Compare comp) {
```

```
3     for (RandomAccessIterator p = first + 1; p < last; ++p) {
4       for (RandomAccessIterator i = p; i > first; --i) {
5         if (comp(*(i - 1), *i)) //array[i] is in right position
6           break;
7         else
8           std::swap(*(i-1), *i);
9       }
10    }
11  }
```

10.3 Lower Bound

It can be proved that all sorting algorithms that just swapping adjacent elements has a lower bound of $\Omega(N^2)$. This is true for both bubble sort and insertion sort.

Chapter 11

Shell Sort

11.1 General Idea

Shell sort tries to break the $\Omega(N^2)$ (the quadratic time barrier) by comparing and swapping elements that are distant from each other. How distant? Shell sort uses a sequence: $h_1, h_2, ..., h_t$ to represent the distance between two elements being compared. h_1 is always equal to 1 and it is the last increment step the Shell sort will use. This means at the end, Shell sort will still sort adjacent elements. However, at that time, many inversions have been solved by previous sorting with increment step $= h_2, h_3, ..., h_t$.

Basically, Shell sort works in this way.

First, we choose an increment sequence: $h_1, h_2, ..., h_t, h_1 = 1$. We start from $h = h_t$. On this phase, we name it as h_t-sort. We compare and sort each following subarray:

$$a_i, a_{i+h_t}, a_{i+2h_t}, ...$$

The index goes to where $i + kh_t$ is within the range. Apparently, $i = 0, 1, 2, ..., h_t - 1$. For every i, we sort the corresponding subarray (using insertion sort, for example. Just treat it like ordinary adjacent array). After we have sorted the subarray for all possible i, the array is said to be h_t-sorted. Elements separated by h_t is in order. We repeat this process until we finish h_1 sort, then the array is sorted. An important property of Shell sort is that if an h_k-sorted array goes through h_{k-1}-sort (i.e. it is h_{k-1}-sorted), it will remain h_k-sorted.

11.2 Implementation

The choice of increment sequence can greatly influence the performance of Shell sort. In this section, Shell's original increment sequence will be used to illustrate the implementation of Shell sort. Shell's increment sequence is:

$$h_t = \frac{\text{size}}{2}, h_{t-1} = \frac{\text{size}}{4}, ..., h_1 = 1$$

We are going to implement a Shell sort routine which has three parameters. Two iterators that give range of the array to sort: `[first, last)`. One function object that with `operator()` defined to give order to two elements in the array: `comp`. The header of the routine is as follows:

```
template <class RandomAccessIterator, class Compare>
void Sort::shellSort(RandomAccessIterator first, RandomAccessIterator last,
↪  Compare comp);
```

We'll use insertion sort to sort each subarray, just treat each h_k apart element as "adjacent". We'll use a for loop to go over all h_k to sort the array:

```
for (int h = (last - first) / 2; h > 0; h /= 2)
```

For each iteration h_k, we start at `*(first + h)` element (or `array[h]`). This is the second element from subarray `array[0]`, `array[0 + h]`, `array[0 + 2h]`, ... This is because the first element in the subarray is naturally sorted (only one element present), so we start with its second element and try to insert it into the ordered array. (Also pay attention that, `array[0]`, `array[1]`, ..., `array[h - 1]` are the first element in each subarray).

To sort all the subarrays, rather than sort one subarray at a time, we can just sort all subarrays simultaneously. Specifically, after we deal with `array[h]`, we move to the next element, `array[h + 1]`, which is the second element from subarray `array[1]`, `array[1 + h]`, `array[1 + 2h]`, ... We insert this element into the corresponding ordered section of its subarray. Then we move on to `array[h + 2]`, and etc, all the way to the last element of the range. We use a for loop to do this:

```
for (auto p = first + h; p != last; ++p)
```

Notice that `p` is an iterator. For each element to be sorted in this for loop (for each `*p`), we perform an insertion operation:

```
// keep value of *p
auto temp = std::move(*p);
// define an iterator to hold where *p should go
auto j = p;
// looking into the sorted section to see where should *p go
for (; j >= first + h && !comp(*(j - h), temp); j -= h)
  *j = std::move(*(j - h));
// after above for loop, j is where *p should go
*j = std::move(temp);
```

In this way, we have sorted the array by Shell sort algorithm. The combined code is as follows:

```
1 template <class RandomAccessIterator, class Compare>
2 void Sort::shellSort(RandomAccessIterator first, RandomAccessIterator last,
  ↪  Compare comp) {
3
4   for (int h = (last - first) / 2; h > 0; h /= 2) {
```

```
5       for (auto p = first + h; p != last; ++p) {
6         auto temp = std::move(*p);
7
8         auto j = p;
9
10        for (; j >= first + h && !comp(*(j - h), temp); j -= h)
11          *j = std::move(*(j - h));
12
13        *j = std::move(temp);
14      }
15    }
16  }
```

11.3 Worst-Case (Shell's increment sequence)

Proof the upper bound of Shell sort using Shell's increment sequence is $O(N^2)$.

We proof this by the following steps:

- calculate the complexity to sort the array for a single pass ($h = h_k$).
- add them together over all passes ($h_t, h_{t-1}, ..., h_1$).

Assume the current increment size is $h_k (h_k < N)$. Then, we have a total of h_k subarrays to be sorted. The first element for each subarray is:

- `array[0]`: first element of subarray 1
- `array[0 + 1]`: first element of subarray 2
- `array[0 + 2]`: first element of subarray 3
- `array[0 + 3]`: first element of subarray 4
- ...
- `array[0 + hk - 1]`: first element of subarray h_k

The total element number is N, the average element number in each subarray is N/h_k. We will perform insertion sort for each subarray. The complexity for insertion sort is $O(n^2)$, where n is the size of array to be sorted. In our case:

$$n = \frac{N}{h_k}$$

So, the complexity for sorting each subarray using insertion sort is:

$$O[(\frac{N}{h_k})^2]$$

As mentioned before, the number of subarrays in this pass is h_k. Thus, total complexity for this pass ($h = h_k$) is:

$$O[h_k \cdot (\frac{N}{h_k})^2] = O(\frac{N^2}{h_k})$$

Now, we can add them togheter over all passes $(h_t, h_{t-1}, ..., h_1)$ to give the total time complexity:

$$O(\sum_{i=1}^{t} \frac{N^2}{h_i}) = O(N^2 \sum_{i=1}^{t} \frac{1}{h_i})$$

For Shell's original increment sequence, $h_{i+1} = 2h_i$, thus:

$$\frac{1}{h_{i+1}} = \frac{1}{2} \cdot \frac{1}{h_i} \Rightarrow \sum_{i=1}^{t} \frac{1}{h_i} = 2 - (\frac{1}{2})^{t-1}) < 2 \text{ (using } h_1 = 1)$$

Thus:

$$O(N^2 \sum_{i=1}^{t} \frac{1}{h_i}) \sim O(N^2)$$

The problem with Shell's original increment is that pairs of increments (h_k, h_{k+1}) are not necessarily relatively prime: they may have common factor. One smaller increment may be the factor of previously larger increment, so it is spending time to sort elements already sorted by the larger increment (notice that checking if elements are sorted also takes time). This causes the smaller increment can have little effect in sorting array.

11.4 Hibbard's increment sequence

In the following discussion, we'll assume sorting the array in ascending order.

Hibbard's increment sequence is as follows:

$$h_k = 2^k - 1$$

For a given array, with N elements, the largest step size h_t should be chosen that is smaller than N. For example, if an array has 9 elements, then the largest increment h_t should be: $2^3 - 1 = 7$.

Now, we focus on calculating the upper bound of Shellsort that sorts an array with N elements using Hibbard's increment sequence. We use the same strategy as before to calculate this:

- calculate the complexity to sort the array for a single pass ($h = h_k$).
- add them together over all passes $(h_t, h_{t-1}, ..., h_1)$.

For a single pass with h_k, we have already proved that the upper bound is $O(\frac{N^2}{h_k})$. This is valid for any sequence you use as long as you use sorting algorithm that has $O(N^2)$ complexity to sort the subarrays. Take insertion sort as an example. During derivation of this result, we have assumed that when sorting each subarray, we have to traverse back to the beginning of the subarray to insert the new item. i.e. this is a worst case. We have to assume this because we don't know the characteristic of our increment sequence $h_1, h_2, ..., h_t$. So we have no idea whether the insertion will stop **BEFORE** reaching the beginning or not. Thus, it is reasonable to make this worst case assumption.

However, since we know the characteristic of Hibbard's increment sequence, we can explore if there is any chance of finding the appropriate place to insert **BEFORE** we reach the beginning of the sorted-subsection. To put it in an Intuitively way, we must take advantage of the fact that the Hibbard's increment sequence is special.

Now, let's take a look at how we can take advantage of the characteristic of Hibbard's increment sequence. First, consider the following case. When we come to h_k-sort the input array, we know that is has already been h_{k+1}-sorted and h_{k+2}-sorted prior to the current h_k-sort. Let's consider elements in position p and $p - i$, where $i \leq p$. We can say:

- if i is a multiple of h_{k+1} or h_{k+2}, then `array[p - i]` and `array[p]` are sorted.
- if i can be linearly expressed by h_{k+1} and h_{k+2}, then `array[p - i]` and `array[p]` are sorted.

The first one is obvious: it is just saying that the distance between `array[p - i]` and `array[p]` is multiples of h_{k+1} or h_{k+2}. So, they must be sorted after h_{k+1}-sort and h_{k+2}-sort. For the second one, we can see the following example.

Imagine we finished 7-sort and 15-sort (for Hibbard's increment, $h_3 = 7, h_4 = 15$), now we are going to do 3-sort ($h_2 = 3$). Let's consider $i = 52$. It can be linearly expressed by 15 and 7: $52 = 1 \times 7 + 3 \times 15$. Let $p = 152$, then we can say that `array[152 - 52]` and `array[152]` are sorted. This is because:

- `array[100]` and `array[100 + 7]` are sorted
- `array[107]` and `array[107 + 15]` are sorted
- `array[122]` and `array[122 + 15]` are sorted
- `array[137]` and `array[137 + 15], or array[152]` are sorted

Generally, if i can be linearyly expressed by h_{k+1} and h_{k+2}, then you can perform the chain conparing shown above using the linear combination and find the sorted pair.

So, if i can be linearly expressed by h_{k+1} and h_{k+2}, then we can expect finding an sorted pair `array[p - i]` and `array[p]`. Once we find an sorted pair, we find a proper position to insert `array[p]` without having to traverse all the way back to the beginning of array.

Now, let's take a look at the characteristic of Hibbard's increment sequence:

$$h_k = 2^k - 1$$

Using this, we have:

$$h_{k+2} = 2h_{k+1} + 1$$

This suggests that h_{k+1} and h_{k+2} are relatively prime—they can't share a common factor. It can be shown that:

> if a and b are relative prime, then a and b can be used as a pair of base to linearly express any integer that is as large as $(a-1)(b-1)$

In our case, we know that if a distance i is larger than $(h_{k+1}-1)(h_{k+2}-1)$, it can be linearly expressed by h_{k+1} and h_{k+2}. (And this is why we only consider two passes prior to h_k, since $(h_{k+1}-1)(h_{k+2}-1)$ is the smallest integer, when we are looking proper position for `array[p]` in the sorted section, i will first reach $(h_{k+1}-1)(h_{k+2}-1)$).

Now plug in $h_{k+1} = 2^{k+1} - 1$ and $h_{k+2} = 2^{k+2} - 1$ into $(h_{k+1}-1)(h_{k+2}-1)$, we have:

$$(h_{k+1}-1)(h_{k+2}-1) = (8h_k + 4)h_k$$

When trying to insert `temp = array[p]` into the sorted section, we are comparing the following paris:

- `<array[p - hk], temp>`
- `<array[p - 2 * hk], temp>`
- `<array[p - 3 * hk], temp>`
- ...
- `<array[p - j * hk], temp>`

If $j = (8h_k + 4)$, $j \cdot h_k = (8h_k + 4)h_k = (h_{k+1}-1)(h_{k+2}-1)$, so it is guaranteed that it can be linearly expressed by h_{k+1} and h_{k+2}. So after we reach this point, we **KNOW** that we don't have to continue to the begining of the sorted section, we can stop at here.

Thus, we need at most $j = 8h_k + 4$ steps to insert an unsorted element into the sorted section of the subarray. And the total number of unsorted elements at h_k-sort is $N - h_k$ (the first h_k elements are sorted in nature, they are the first elements of each h_k subarray). Thus, for each pass h_k, the bound should be:

$$(8h_k + 4)(N - h_k) \sim O(Nh_k)$$

Is this bound suitable for all passes from h_t to h_1, so the total bound is $\sum_{i=1}^{t} O(Nh_i)$?

Actually, not really. Intuitively, we can think that, as h_i becomes large, $(8h_i + 4)h_i$ will become so large, that `a[p - hi * (8 * hi + 4)]` goes beyond the beginning position, i.e. you will reach beginning before you reach a point whose distance bwteen `*p` can be linearyly expressed by $(h_{k+1} - 1)(h_{k+2} - 1)$. Our question becomes, how large h_k should bem if the "linear-combination sorted" case can be reached earlier than the worst-case of general insertion sort (reaching the begining)?

To answer this question, let's consider an array of size N. If $N = 2^t$, then $h_t = 2^t - 1$ is the largest step size in the increment sequence. The complexity of insertion sort is:

$$O(\frac{N^2}{h_k})$$

The complexity of "linear combination sorted" case is:

$$O(Nh_k)$$

Let:

$$\frac{N^2}{h_k} = Nh_k$$

solve it, we have:

$$h_k = N^{\frac{1}{2}}$$

This means, when $h_k < N^{\frac{1}{2}}$, the "linear combination" case will apprear first before reaching to the beginning of the array. When $h_k < N^{\frac{1}{2}}$, original insertion sort will be faster to reach the begining. Plug in $N = 2^t$ into $h_k = N^{\frac{1}{2}}$:

$$h_k = 2^{\frac{t}{2}} \sim 2^{\frac{t}{2}} - 1 \sim \text{roughly } k = \frac{t}{2}$$

Thus, roughly, about half of the increments (h_ks) satisfy $h_k < N^{\frac{1}{2}}$.

So, we have to calculate the upper bound in two sections:

1. $h_k \leq N^{\frac{1}{2}}$. In this section, the upper bound is $O(Nh_k)$
2. $h_k > N^{\frac{1}{2}}$. In this section, the upper bound is $O(\frac{N^2}{h_k})$

Assume t is even, then we have:

$$O(\sum_{k=1}^{\frac{t}{2}} Nh_k + \sum_{k=\frac{t}{2}+1}^{t} \frac{N^2}{h_k}) = O(N\sum_{k=1}^{\frac{t}{2}} h_k + N^2 \sum_{k=\frac{t}{2}+1}^{t} \frac{1}{h_k})$$

Pay attention that, both sums are geometric series (the latter is semi-geometric). So in the first sum, the last term dominates (because h_t is the largest), so:

$$O(N\sum_{k=1}^{\frac{t}{2}} h_k) = O(Nh_{\frac{t}{2}})$$

In the second sum, the first term dominates (because $h_{\frac{t}{2}}$ is the smallest, so $\frac{1}{h_{\frac{t}{2}}}$ is the largest), so we have:

$$O(N^2 \sum_{k=\frac{t}{2}+1}^{t} \frac{1}{h_k}) = O(\frac{N^2}{h_{\frac{t}{2}}})$$

The combined upper bound is :

$$O(Nh_{\frac{t}{2}}) + O(\frac{N^2}{h_{\frac{t}{2}}})$$

Since $h_{\frac{t}{2}} = 2^{\frac{t}{2}} - 1 \sim N^{\frac{1}{2}} \Rightarrow h_{\frac{t}{2}} = \Theta(N^{\frac{1}{2}})$, we have:

$$O(Nh_{\frac{t}{2}}) + O(\frac{N^2}{h_{\frac{t}{2}}}) = O(N^{\frac{3}{2}})$$

The upper bound of Shell sort, using Hibbard's increments, is $O(N^{\frac{3}{2}})$.

Chapter 12

Heap Sort

12.1 General Idea

For an unordered array, we can first build heap order in it (by the `buildHeap()` routine). This step takes $O(N)$ time. Then, we can perform N `deleteMin()` operations to extract the elements in the array in order. For each `deleteMin()`, time required is $O(\log N)$. So the total time required to "delete" all elements is $O(N \log N)$. We can move the deleted item into the last slot of the current array (where hole is created, of course, we still have to `currentSize--`). In this way, we can sort the array in place, no need of extra memory space to hold the temporary array.

12.2 Implementation

12.2.1 Analysis

In this example, we assume the internal complete binary tree starts at `array[0]`. The header for my `heapSort()` function is as follows (which is part of `Sort` class):

```
1 template <class Iterator, class Compare>
2 void heapSort(Iterator first, Iterator last, Compare comp);
```

This function will sort the array in the range defined by `[first, last)`. The object `comp` will be used to give order to the elements in the range. This object is assumed to have operator() (the function call operator) overloaded to give proper ordering information.

We implement the routine in the following steps:

- build up a heap order in the range. We'll use `comp` to determine the relative order of the elements: to determine which will be at the root.
- take away element at the root (the smallest or biggest element at the time of being deleted from the original array, this operation will create a hole in root). You also have

to move the element at the last slot to the hole. Take together, what you should do is `std::swap(array[root], array[currentSize-1])`.

- `percolateDown()` the root (to restore the heap order).
- continue these steps until you traversed the range, sorting complete.

12.2.2 Build up heap order

Before diving into details, one thing must be made clear. From the function header, we know that we want to **SORT** the range by the logic defined by `comp`. However, as we can see from the mechanism of heap sort, the actual sequence is the inverse of the target sequence. In order to address this issue, we can build a heap order that is the inverse of the order defined by `comp`. In the following section, we'll apply this idea.

To build up heap order, we start from the first node that has child in the range, percolate down one by one until we percolate down the root. The number of node that has at least one child in a heap of size N with root at `array[1]` is $N/2$ (integer division). For a heap of size N with root at `array[0]` is $N/2-1$. So, we start from `array[N/2-1]`, all the way until we percolate down `array[0]`.

```
//get the size of the range
int currentSize = last - first;
for (int i = currentSize/2-1; i >= 0; --i) {
  percolateDown(first, last, comp, i);
}
```

We used a routine `percolateDown()` in the above code. What it does is to check if element at `*(first + i)` is violating the heap property (i.e. greater or smaller than its child or children, depending on what `comp` is). If it violates, then We'll move the child to its position and continue checking if it violating heap property again (with its new child). We continue this process until:

- we find a proper position that it does not violate the heap property. Or:
- the hole reached bottom (i.e. there is no child, can't go further down)

We declare the `percolateDown()` routine as follows:

```
template <class Iterator, class Compare>
void percolateDown(Iterator first, Iterator class, Compare comp, int hole )
 ↪  {
  int child;
  int currentSize;

  auto tmp = std::move(*(first + hole));

  for (; hole * 2 + 1 <= currentSize; hole = child) {
    child = hole * 2 + 1; //left child
```

```
10
11      // check right child's existence
12      // if it exists, pick the child according to comp
13      // pay attention that the order is the inverse of what comp() defined
14      if (child != currentSize && comp(*(first + child), *(first + child +
        ↪  1)))
15        ++child;
16
17      //if the value in hole violated heap property
18      //percolate hole down
19      if (comp(tmp, *(first + child)))
20        *(first + hole) = std::move(*(first + child));
21      else //heap property not violated, proper position found
22        break;
23    }
24
25    *(first + hole) = std::move(tmp);
26  }
```

12.2.3 Sort Using Heap Order

We need a routine to take away the element at the root and re-build the heap. This is similar with `deleteMin()` in binary heap class. The difference is that we keep the "deleted" element to build up a sorted array. The working steps are as follows:

- define an integer, `currentSize`, to hold the number of unsorted elements. Pay attention that we don't need to move the last element (its already in sorted position).
- use a for loop to operate `currentSize - 1` times. (begin at `currentSize = last - first`, stop at `currentSize = 1`):
  - swap the first and last item in the range (`std::swap(*(first), *(first + currentSize - 1))`). Now the original root is in sorted section.
  - `percolateDown()` the first item (which now contains the previously last item) to proper position. This is to restore heap property. The range you pass into `percolateDown()` should be the shrinked one, because you have to assume the heap size is reduced by one.
  - decrement `currentSize` by 1.

Code for `heapSort()`:

```
1  void Sort::heapSort(RandomAccessIterator first, RandomAccessIterator last,
   ↪  Compare comp) {
2    //first step: build up heap order
3    buildHeap(first, last, comp);
4
```

```
5     //begin sorting
6     int currentSize = last - first;
7     for (; currentSize > 1; --currentSize) {
8       std::swap(*(first), *(first + currentSize - 1));
9       if (currentSize > 2)
10        //hole is always the first element
11        percolateDown(first, first + currentSize - 2, comp, 0);
12    }
13  }
```

12.3 Complexity Analysis

(this section may be flawed, it is based on my calculation. I couldn't understand the textbook's derivation.) The heap sort contains two parts: (1) create heap order (heapify) inside the target range; (2) sort the array by repeatedly take the root of the heap (the first term in the array) to build up an ordered array.

12.3.1 Build Heap

To build heap order in a random array, we start from the first node that has at least one child. We check if this node violates the heap order. If so, we swap it with one of its child. For a heap with height h, the total height is $2^{h+1}-(h+1)-1$. This value is the total number of layers to percolate down for all the nodes (the worst case). On the other hand, the height of heap can be expressed by the number in the heap by: $h = \log(N+1) - 1$. Plug into the expression of total height, the number of operations is: $N - \log(N+1)$.

12.3.2 Sort

We assume the worst case, during every time of the iteration, we have to percolate the root down to bottom. Percolating down each layer requires two operations: (1) comparing the children of the node and determine which child to be used to compare with the node; (2) compare the chosen child with the node and determine if we have to swap the two. As analyzed before, for an array of size N, we only need to sort the first $(N-1)$ elements. For ith iteration, we moved i elements to sorted section, and there are $N-i$ elements unsorted. These $N-i$ elements composed the remaining heap. Now we have to percolate down the element at root position all the way to bottom (for the worst case). For a **FULL** complete binary tree with $N-i$ elements, the relation between height h and number of elements are: $2^{h+1}-1 = N-i$. Thus: $h = \log(N-i+1) - 1$. In reality, the complete binary tree may not be full, so the h value calculated by the above expression is smaller than the worst case. We can fix that by

$$h = 1 + \lfloor \log(N-i+1) - 1 \rfloor \ = \lfloor \log(N-i+1) \rfloor$$

Thus, for ith iteration (i.e. after moving i elements to the sorted section), the number of operations involved to percolate root down to bottom (the worst case) is:$\lfloor \log(N-i+1) \rfloor$. Each percolating down contains 2 operations, so total operations are: $2\lfloor \log(N-i+1) \rfloor$.

i is from 1 to $N-1$. Thus we sum all the operations:

$$\sum_{i=1}^{N-1} \lfloor 2\log(N-i+1) \rfloor$$

(To be continued...)

Chapter 13

Merge Sort

13.1 General Idea

The foundamental operation in this sorting algorithm is merging two sorted lists `A` and `B`. We use another chunk of memory (`C`) which has the size of `A.size() + B.size()`. And we have three iterators (`a`, `b` and `c`) pointing to the beginning of `A`, `B` and `C`, respectively. We now start to merge `A` and `B` into a single array `C`. For each element we insert to `C`, We compare the element pointed by the `a` and `b`, and pick the one that satisfy the order into where `c` is pointing at, then we increment the iterator (which was pointing to the inserted element in `A` or `B`) and `c`, and begin next inserting. If `a` or `b` reaches the end, we inserting the remaining elements in the non-empty array into `c` in its original sequence. After these operations, we have obtained a sorted array `C`.

Merge sort is a fine example of using divide and conquer strategy (with recursive algorithm). For a given array, we can use merge sort to sort its first half, then its second half, then we merge it into one array and copy back to the original array. The problem is *divided* into smaller problems and solved recursively. The *conquering* phase consists of patching together the answers.

13.2 Implementation

I'll implement the merge sort which accepts three parameters: two iterators that mark the range of the array to be sorted, and one function object that provides a routine to determine the relative order of elements in the array (with `operator()` defined). The merge sort function will sort the array marked in `[first, last)`. The header is as follows:

```
1 template <class RandomAccessIterator, class Compare>
2 void Sort::mergeSort(RandomAccessIterator first, RandomAccessIterator last,
  ↪  Compare comp);
```

During the sorting, we have to use an extra space to hold the temporary sorted array. If we declare an array in each recursive call of merge sort, it would be a huge cost, roughly

$\log N$ in total. Notice that, at any given moment, we only use one temporary array to hold the result (because we'll only proceed to sort next sub-array once we finished current one). This means we can declare an array just for this purpose and pass it into internal private implementation of merge sort. The above header then describes the public routine to be called from outside. Namely:

```
template <class RandomAccessIterator, class Compare>
void Sort::mergeSort(RandomAccessIterator first, RandomAccessIterator last,
 ↪  Compare comp) {
  //base case: only one element in the array
  if (first + 1 == last)
    return;

  //as long as element number >= 2, we need to sort
  typename std::vector<typename
   ↪  std::remove_reference<decltype(*first)>::type> sorted_array(last -
   ↪  first);// this vector will be used to hold intermediate merged array
   ↪  for all recursively called function

  //call private version of mergeSort to finish the work
  mergeSort(first, last, comp, sorted_array);

}
```

By the way, the above code shows how to use a generic iterator to create a vector that can hold the same type as `*iterator`.

The base case for merge sort recursive function is `first + 1 == last`. This means the passed in array has only one element, so it is "naturally" sorted. In this case, there is nothing to be done, just return.

We devide the array into two parts:

- `[first, first + (last - first) / 2)`
- `[first + (last - first) / 2, last)`

Then, we call merge sort to sort these two parts, then we merge them together (using the passed in array as the temporary storage space).

The code for the internal private version of merge sort is as follows:

```
template <class RandomAccessIterator, class Compare, class Element>
void Sort::mergeSort(RandomAccessIterator first, RandomAccessIterator last,
 ↪  Compare comp, std::vector<Element>& sorted_array) {
  // base case
  if (first + 1 == last)
    return;

```

```
7     // calculate the size of the array
8     int size = last - first;
9
10    // recursively call itself to sort the first half
11    mergeSort(first, first + size / 2, comp, sorted_array);
12
13    // recursively call itself to sort the second half
14    mergeSort(first + size / 2, last, comp, sorted_array);
15
16    // merge the sorted sub-array
17    auto itr_1 = first; // begin of sub-array_1
18    auto itr_2 = first + size / 2; // begin of sub-array_2
19    auto itr_array = sorted_array.begin(); // begin of temporary array
20
21    while (itr_1 != first + size / 2 && itr_2 != last) {
22      if (comp(*itr_1, *itr_2))
23        *(itr_array++) = *(itr_1++);
24
25      else
26        *(itr_array++) = *(itr_2++);
27    }
28
29    // after the while loop, one itr reached end
30    // dump remaining sorted element into temporary container
31    if (itr_1 == first + size / 2)
32      while (itr_2 != last)
33        *(itr_array++) = *(itr_2++);
34
35    else
36      while (itr_1 != first + size / 2)
37        *(itr_array++) = *(itr_1++);
38
39    // copy back sorted array
40    for (itr_1 = first, itr_array = sorted_array.begin(); itr_1 != last; ) {
41      *(itr_1++) = *(itr_array++);
42    }
43  }
```

Chapter 14

Quick Sort

14.1 General Idea

Average running time: $O(N \log N)$.

Worst-case performance: $O(N^2)$.

Quick sort uses same strategy as merge sort: divide and conquer recursive algorithm. The general idea is this (assume we want sort the array in increasing order). For a given input array, we pick one item in the array, named as `pivot`. We divide the array into three parts:

- subarray `s1`, which contains all elements that are smaller than `pivot`
- the `pivot`
- subarray `s2`, which contains all elements that are larger than `pivot`

The problem has been divided into two sub-problems: sort `s1` and `s2`. So, we call quick sort again to deal with `s1` and `s2`. After `s1` and `s2` are sorted, the whole array has been sorted.

The process of dividing the array according to `pivot` is called **partition**. During the partition, we may encounter some elements that are equal with `pivot`. How to partition these elements remained an implementation issue. Intuitively, we would hope that about half of these duplicated elements go into `s1` and the other half go into `s2`, so the two sub-arrays are somewhat "balanced".

Two differences between quick sort and merge sort are:

1. quick sort divide the array by pivot value, while merge sort divide the array purely by the middle point.
2. quick sort doesn't need extra space to **merge** the sorted subarray into final sorted array, sorting is done in place, while merge sort needs.

The details of how we pick the `pivot` and how we partition the array can greatly influence the performance of quick sort. There are many ways to do it, in this section, a widely accepted

method will be explored. As before, we'll use sorting in increasing order as an example to illustrate.

14.2 Picking the Pivot

For a given array, we find the median of the following three elements:

1. the first element in array
2. the center element in array
3. the last element in array

14.3 Partition Strategy

After finding the `pivot` element, we use following steps to partition the array.

1. get the pivot element out of the way by swapping it with the last element.
2. define two iterators `i` and `j`. `i` will be pointing to the first element in the array. `j` will be pointing to the next-to-last element (i.e. the first element before the `pivot`). The idea is to put elements that are smaller than the `pivot` at the front of the array, and elements that are bigger than the `pivot` at the back of the array.
3. `i` start to move toward the end, `j` start to move toward the beginning. `i` and `j` kept moving until they encounter elements that are not supposed to be there—i.e. `i` has met element that is bigger than `pivot`, `j` has met element that is smaller than `pivot`:

   ```
   while (*i < pivot)
     ++i;
   while (*j > pivot)
     ++j;
   ```

   After the above loop, both `i` and `j` are pointing to an element. At this moment, if `i` is still before `j`, we should swap the elements pointed by `i` and `j`, and increment `i`, `j`:

   ```
   if (i < j)
     std::swap(*(i++), *(j--));
   ```

4. we keep repeating step 3 until `i` and `j` crossed each other, i.e. `i >= j`.
5. after `i` and `j` crossed each other, we swap `pivot` and `*i`. Pay attention that now `i` is pointing to the first element that is larger than `pivot`. After we do this swapping, the array is in following structure:

   ```
   --- pivot ++++++
   ```

   where `-` represents elements that are smaller than `pivot`; `+` represents elements that are greater than `pivot`.

14.4 Small Arrays

For very small arrays, quick sort does not perform as well as insertion sort. Furthermore, because quick sort is recursive, these cases will occur eventually, even if you are sorting a big array. A common solution is not to use quick sort recursively for small arrays, but instead use a sorting algorithm that is efficient for small arrays. A good cutoff is `N == 10`.

14.5 Implementations

Implementation of pivot() function:

```
template <class RandomAccessIterator, class Compare>
void Sort::pivot(RandomAccessIterator first, RandomAccessIterator last,
 ↪  Compare comp) {
  auto center = first + (last - first) / 2;

  //rearrange elements at first, center and (last - 1) in order
  if (!comp(*first, *center))
    std::swap(*first, *center);
  if (!comp(*first, *(last - 1)))
    std::swap(*first, *(last - 1));
  if (!comp(*center, *(last - 1)))
    std::swap(*center, *(last - 1));

  //after above step, center is pointing to median of the three position
  //move pivot to the slot before the last slot
  //this is because pivot and the last slot is already in order
  std::swap(*center, *(last - 2));
}
```

Implementation of quick sort:

```
template <class RandomAccessIterator, class Compare>
void Sort::quickSort(RandomAccessIterator first, RandomAccessIterator last,
 ↪  Compare comp) {
  //base case: small array
  if (last - first < 11) {
    insertionSort(first, last, comp);
    return;
  }


  //call a private routine to select pivot and move it to the slot before
   ↪  last
  //this is to prepare for partition
  pivot(first, last, comp);
```

```
13
14    //begin partition
15    //notice that, *(last - 2) is the pivot value
16    auto i = first;
17    auto j = last - 3;
18
19    while (i < j) {//begin moving i and j
20      while (comp(*i, *(last - 2)))
21        ++i;
22      while (!comp(*j, *(last - 2)))
23        --j;
24
25      if (i < j) //if i j not crossed yet
26        std::swap(*(i++), *(j--));
27    }
28
29    //after the above loop, i and j crossed
30    //swap *i and pivot to finish partition
31    std::swap(*i, *(last - 2));
32
33    //partition finished, now the loop has the structure:
34    //----- pivot ++++++ or +++++ pivot ------
35    //next step is to call quick sort recursively to sort two sub-arrays
36    quickSort(first, i, comp);
37    quickSort(i + 1, last, comp);
38  }
```

Part IV

Gimmicks

Chapter 15

C++ Related

15.1 Create a Vector Using its Iterator

Demand first encountered: when trying to write iterator implementation of merge sort. I have to define a generic vector, using its iterator. After some search on internet, I found the following tricks to do this:

```
std::vector<int> v1;
auto itr = v1.begin();
std::vector<typename std::remove_reference<decltype(*itr)>::type> v2;
// after the above declaration, v2 is declared as a vector holding integer
↪  type
```

Reference: Create a vector using its iterator