CS335 Class Notes

Luis F. Uceta

2020-01-28

Contents

1 Lecture 1: C++ basics

1.1 C++ classes

```
class IntCell {
    public:
        // This is a constructor
        IntCell() { _value = 0; }

        // This is another constructor. It takes an initial value.
        IntCell( int initialValue ) { _value = initialValue; }

        // This is an accessor (or getter) method.
        int read() { return _value; }

        // This is a setter method.
        void write( int newValue ) { _value = newValue; }

    private:
        int _value;
};

int main() {
    // Using the first constructor w/o parameters
    IntCell x;
    int xValue = x.read(); // 0
    x.write(5);
    xValue = x.read();    // 5

    // Using the first constructor with a single parameter
    IntCell y(12);
    int yValue = y.read(); // 12
    y.write(y.read() * 2);
    yValue = y.read();    // 24
}
```

The keywords **`public`** and **`private`** determine the visibility of class members. A member that is **`public`** may be accessed by any method in any class while a member that is **`private`** may only be accessed by methods in the class it's declared. Date members are usually declared **`private`** to hide a class's internal details (i.e., **information hiding**).

1.2 Initialization list

Instead of initializing data members inside a constructor's body, an **initialization list** can be used to do so right in the constructor's signature:

```
class IntCell {
    public:
        explicit IntCell( int initialValue = 0 ) : _value { initialValue }
            { }
        int read() const { return _value; }
        void write( int newValue ) { _value = newValue; }

    private:
        int _value;
};
```

By default C++ does behind-the-scenes type conversions in all one-parameter constructors. Thus, to avoid this the keyword `explicit` is used.

With **implicit type conversion**,

```
IntCell obj;
obj = 37;
```

means

```
IntCell temp = 37;
obj = temp;
```

With **explicit type conversion** (by using `explicit`), a "type mismatch" compiler error is thrown instead. The use of `explicit` means a one-parameter constructor cannot be used to generate an implicit temporary object.

1.3 Constant member functions

A member function that examines but does not change the state of its object is an **accessor**. In the `IntCell`, `read` is an accessor and thus it's marked explicitly as an accessor by using the keyword **`const`** after the closing parenthesis that ends the parameter list. This signals that this method doesn't change the state of an `IntCell` and if it tries the compiler complains.

1.4 Interface and implementation

An **interface** lists the class and its members (both data and methods) and the **implementation** provides the implementations of the methods.

In C++, this is done by placing the interface in a `.h` file and the implementation in a `.cpp` file. Source code that requires knowledge of the interface must include the interface file (e.g., `#include Interf`

.h). To avoid including files multiple times, a few preprocessor commands are used in the interface file:

```
#ifndef INTCELL_H // if INTCELL_H is not defined...
#define INTCELL_H // ...define it.

// Here's the IntCell class's interface

#endif            // end the definition
```

1.4.1 IntCell interface

```
// IntCell.h
#ifndef INTCELL_H
#define INTCELL_H

class IntCell {
    public:
        explicit IntCell( int initialValue = 0 ) {}
        int read() const { return _value; }
        void write( int newValue ) { _value = newValue; }

    private:
        int _value;
};

#endif
```

1.4.2 IntCell implementation

```
// IntCell.cpp
#include "IntCell.h"  // we're making use IntCell's interf. so we include
    it.

IntCell::IntCell( int initialValue = ) : _value { initialValue } { }

// Return the stored value.
int IntCell::read() const { return _value; }

// Change the stored value.
void IntCell::write( int newValue ) { _value = newValue; }

};
```

Notice that we must use the **scope resolution operator** (`::`) to identify the class each member function belongs to. Otherwise, it's assumed that the function is in the **global scope**. Also note that the signature of an implemented member function *must* match exactly the signature in the class interface.

1.4.3 Using IntCell

```
// main.cpp
#include "IntCell.h"

int main() {
    IntCell m{};
    m.write(5);
    int val = m.read(); // 5

    return 0;
}
```

To compile the program, run:

```
g++ -o prog main.cpp IntCell.cpp
```

The header files (e.g., `IntCell.h`) aren't listed since they're included in the implementation files and thus are compiled too.

1.5 C++ Objects

In C++ objects are declared much like primitive types (e.g., **`char`**, **`int`**, **`float`**, etc.):

```
IntCell obj1;     // Zero parameter constructor
IntCell obj2(12); // One parameter constructor
```

However, due to an effort to standardize the uniform initialization syntax using braces, it's recommended to declare objects as follows:

```
IntCell obj1;     // Zero parameter constructor
IntCell obj1{};   // same as before
IntCell obj2{12}; // One parameter constructor
```

1.6 `vectors` and `strings`

The C++ standard defines the classes `vector` and `string` that intend to replace the built-in C++ array. Unlike arrays, these classes behave like first-class objects. Thus, they can be copied with =, they remember how many items they can store, and their indexing operator check that the provided valid (to access some element) is valid.

1.6.1 `vector`

```
vector<int> numbers = {1, 2, 3, 4, 5}; // Or...
vector<int> numbers {1, 2, 3, 4, 5};
```

However, there's some ambiguity with the declaration. For instance, `vector<int> a(12)` declares a vector that stores 12 integers but `vector<int> a{12}` declares a vector of size 1 with the value 12 at position 0. This is because C++ gives precedence to the initializer list. If the intention is to declare a vector of size 12, use the old C++ syntax using parentheses: `vector<int> a(12)`.

Instead of using the regular for loop, vectors can be looped over using the following syntax:

```
int sum = 0;
vector<int> numbers = {1, 2, 3, 4, 5};
for( int num : numbers ) {
    sum += num;
}
```

The keyword `auto` can be used to let the compiler determine the type:

```
int sum = 0;
vector<int> numbers = {1, 2, 3, 4, 5};
for( int num : numbers ) {
    sum += num;
}
```

However, keep in mind that this range syntax is only appropriate when 1) accessing elements sequentially and 2) when the index isn't needed.

1.7 C++ details

1.7.1 Pointers

A **pointer** is a variable that stores the address of a memory location (i.e., where an object resides). The syntax for declaring a pointer is as follows:

```
char* ptr_c;   // ptr_c is a pointer-to-char
int *ptr_i;    // ptr_i is a pointer-to-int
float * ptr_f;  // ptr_f is a pointer-to-float
void *u_ptr;   // ptr_u is a untyped pointer
```

Note that the position of the asterisk `*` (known as the **indirection operator** in this context) doesn't matter.

In order to obtain an object's address, the **address-of** operator & is used and a pointer variable is used to store it:

```
int num = 5;
int *ptr_num = &num; // get the address of num
int val = *ptr_num;  // get the value stored in the address ptr_num points
    to
*ptr_num = 6;        // now num is also 6. after all ptr_num to the same
   memory location
```

Thus, the indirection operator has two jobs: 1) it declares a pointer and 2) it dereferences a pointer.

1.7.2 Dynamic object allocation and garbage collection

Objects are created dynamically by allocating memory with **new** which returns a pointer to the newly created object:

```
IntCell *m;
m = new IntCell(); // OK
m = new IntCell{}; // OK
m = new IntCell;   // OK
```

If a pointer variable points at a class type, then a (visible) member of the object being pointed at can be accessed via the -> operator:

```
int x = m->read();
m->write(6);
```

When an object that is allocated by **new** is no longer referenced, the `delete` operation must be applied to the object (through a pointer). Otherwise, the memory that it consumes is lost (until the program terminates). This is known as a **memory leak**.

```
delete m;
```

1.7.3 Lvalues, and Rvalues

An **lvalue** is an expression that identifies a non-temporary object:

```
vector<int> arr(3);
const int x = 2;
int y;
int z = x + y;
vector<string> *ptr = &arr;
```

The expressions `arr`, `x`, `y`, `z`, `ptr`, `*ptr`, and `z` are all lvalues.

An **rvalue** is an expression that identifies 1) a temporary object or 2) a value (e.g., literal constant) not associated with any object:

```
vector<int> arr(3);
const int x = 2;
int y;
int z = x + y;
vector<string> *ptr = &arr;
```

Here, `2`, `x+y`, and `&arr` are all rvalues.

1.7.4 References

A reference type allows us to define a new name for an existing value. In classic C++, a reference can generally only be a name for an lvalue, since having a reference to a temporary would lead to the ability to access an object that has theoretically been declared as no longer needed, and thus may have had its resources reclaimed for another object. However, in C++11, we can two types of references:

- An **lvalue reference** is declared by placing an & after some type. An lvalue reference then becomes a synonym for the object it references.

```
string str = "hello";
string &str_r = str;
str_r += " world"; // now str is "hello world"

string &lit_r = "hi";      // ILLEGAL: "hi" is not modifiable
string &bad_r = str + "!"; // ILLEGAL: str + "!" isn't an lvalue
```

- An **rvalue reference** is declared by placing an && after some type. An rvalue reference has the same characteristics as an lvalue reference except that, unlike an lvalue reference, *an rvalue reference can also reference an rvalue* (i.e., a temporary).

```
string str = "hell";
string && bad1 = "hello";            // LEGAL
string && bad2 = str + "";           // LEGAL
string && sub  = str.substr( 0, 4 ); // LEGAL
```

1.7.5 Uses of lvalue references

- **Aliasing complicated names.** We can rename objects that are too long and complicated to simpler names.

```
auto &whichList = theLists[ myhash( x, theLists.size( ) ) ];
if( find( begin( whichList ), end( whichList ), x ) != end( whichList ) )
    {
     return false;
}
whichList.push_back( x );
```

Note that simply writing `auto whichList = theLists[ myhash( x, theLists.size( ))];` wouldn't work because this would create a copy, and `whichList.push_back( x );` would be applied to the copy, not the original.

- **Range `for` loops.** By default, a range `for` loop cannot change the elements it iterates over, however taking a lvalue reference allows to modify those elements.

```
vector<int> numbers = {1, 2, 3};

for( auto &number : numbers ) {
    number++;
}
```

- **Avoiding a copy.** Given a function that returns an element of an array/vector, we could return a non-modifiable reference to that element instead of a copy. For instance, instead of

```
// @arr: A non-empty vector of strings.
// @return the maximum string in the @arr.
//  Will abort() if @arris empty.
string FindMax1(constvector<string>&arr) {
    if (arr.empty()) abort();
    int max_index=0;
    for (inti =1; i < arr.size(); ++i) {
        if(arr[max_index]<arr[i]) { max_index = i; }
    }
    return arr[max_index];
}
```

we could have

```
// @arr: A non-empty vector of strings.
// @return the maximum string in the @arr.
//  Will abort() if @arris empty.
const string &FindMax1(constvector<string>&arr) {
    if (arr.empty()) abort();
    int max_index=0;
    for (inti =1; i < arr.size(); ++i) {
        if(arr[max_index]<arr[i]) { max_index = i; }
    }
    return arr[max_index];
}
```

Syntax is needed in function declarations and returns to enable the passing and returning using references instead of copies. Notice the `const` keyword and & in the function's header.

2 Lecture 2: C++ Basics (cont.)

2.1 Parameter passing

Many languages, including C++, pass all parameters using **call-by-value**: the actual argument is copied into the formal parameter. However, this might be utterly inefficient if large complex objects are being passed since they're copied in their entirety. Historically C++ has had three ways to pass parameters:

- **call-by-value:** Useful to pass small objects that shouldn't be mutated by the function.

```
double average( double a, double b );
double x = 2.5, y = 3.5;
double z = average(x, y); // x and y remain unchanged

void swap( double a, double b ); // swap a and b
swap(x, y); // Not what's expected; x and y remain unchanged
```

- **call-by-reference (call-by-lvalue-reference):** Useful for all type of objects that may be changed by the function.

```
double x = 2.5, y = 3.5;
swap( &a, &b );
swap(x, y); // now x = 3.5 and y = 2.5
```

- **call-by-constant-reference:** Useful for large objects that are expensive to copy and that must not be changed by the function. For this, the parameter is declared as a reference and the keyword `const` is used to signal that it cannot be modified.

```
string randomItem( const vector<string> &arr ); // return a random item in
    arr
```

There's still another fourth way to pass parameters:

- **call-by-rvalue-reference:** Instead of copying a temporary object stored in an rvalue, a move is used; moving an object's state is easier than copying, as *it may involve just a simpler pointer change*. Functions know if a value is temporary or not based on their signature so the primary use for this type of parameter passing is overloading a function based on whether a parameter is an lvalue or rvalue.

```
string randomItem( const vector<string>  &arr ); // return random item in
    lvalue arr (single &)
string randomItem(       vector<string> &&arr ); // return random item in
    rvalue arr (double &)

vector<string> v = {"hello", "world"};
string x = randomItem(v);                  // invoke lvalue function
string x = randomItem({"hello", "world"}); // invoke rvalue function
```

This idiom is particularly useful for defining the behavor of = and in writing constructors.

2.2 Return passing

In C++, there are several mechanism for returning from a function:

- **return-by-value:** A copy of the object is returned which can be inefficient, however in C++11 return-by-value may still be efficient for large objects if the returned objects are rvalues.

```
double average( double a, double b );
LargeType randomItem( const vector<LargeType> &arr ); // potentially
    inefficient
vector<int> partialSum( const vector<int> &arr );     // efficient in C
    ++11
```

The following are two versions of the function `randomItem`. The second version avoids the creation of a temporary `LargeType` object, but only if the caller accesses it with a constant reference:

```
LargeType randomItem1( const vector<LargeType> &arr ) {
    return arr[ randomInt(0, arr.size() -1) ];
}

const LargeType & randomItem2( const vector<LargeType> &arr ) {
    return arr[ randomInt(0, arr.size() -1) ];
}

vector<LargeType> vec;
LargeType item1 = randomItem2(vec);        // copy
LargeType item2 = randomItem2(vec);        // copy
const LargeType &item3 = randomItem2(vec); // no copy
auto &item4 = randomItem2(vec);            // no copy
```

- **return-by-constant-reference:** This avoid creating an immediate copy of an object. However, the caller must also use a constant reference to access the return value; otherwise, there will be still a copy. What does the constant reference mean? It means that we don't want to allow changes to be made by the caller by using the return value.

```
const LargeType & randomItem2( const vector<LargeType> &arr ) {
    return arr[ randomInt(0, arr.size() -1) ];
}

vector<LargeType> vec;
const LargeType &item3 = randomItem2(vec); // no copy
```

- **return-by-reference:** A reference is returned and the caller can modify the returned value. This is used in a few places to allow the caller of a function to have modifiable access to the internal data representation of a class.

2.3 C++11 `std::swap` and `std::move`

Given that copying large objects is expensive, C++11 allows the programmer to easily replace expensive copies with moves provided the object's class supports moves.

Take the following example of a `swap` function that swap its arguments by three copies:

```
void swap( vector<string> &x, vector<string> &y ) {
    vector<string> tmp = x;
    x = y;
    y = tmp;
}
```

In C++11, if the right-hand side of the assignment operator (or constructor) is an rvalue, then if the object supports moving, we can automatically avoid copies. In the example above, we know that `vector` supports moving so instead of copy operations we could do move operations. These could be done either by casting the right-hand side of an assignment to an rvalue reference or by using `std::move`.

```
// Using type-casting
void swap( vector<string> &x, vector<string> &y ) {
    vector<string> tmp = static_cast<vector<string> &&>( x );
    x = static_cast<vector<string> &&>( y );
    y = static_cast<vector<string> &&>( tmp );
}

// Using std::move, equivalent to casting but more succint
void swap( vector<string> &x, vector<string> &y ) {
    vector<string> tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
}
```

NOTE: `std::move` doesn't move anything; rather, it makes a value (either lvalue or rvalue) subject *to*

be moved.

It's worth noting that `std::swap` is already part of STL and works for any part:

```
vector<string> x;
vector<string> y;
std::swap(x, y); // x contains y's contents and y contains x's contents
```

2.4 The big five

In C++, classes come with five special functions already written for each class. These are the **destructor**, **copy constructor**, **move constructor**, **copy assignment operator**, and **move assignment operator**. In many cases, you accept the default behavior provided by the compiler can be relied on. Sometimes you cannot.

Let's assume the following interface for the `IntCell` class:

```
#ifndef INTCELL_H
#define INTCELL_H

class IntCell {
    public:
        explicit IntCell( int initialValue = 0 ) { value_ = new int{
            initialValue}; }
        int read() const {}
        void write( int x ) {}
    private:
        int *value_;
};

#endif
```

2.4.1 Destructor

This function is called whenever an object goes out of scope or is subjected to a `delete` operation. Typically, the only responsibility of the destructor is to free up any resources that were acquired during the use of the object. This includes calling `delete` for any corresponding **`new`**s, closing any files that were opened, and so on.

```
IntCell::~IntCell() {
    delete value_;
}
```

2.4.2 Copy constructor and move constructor

These two constructors are required to construct a new object, initialized to the same state as another object of the same type.

- If the existing object is an **lvalue**, then it's a **copy constructor**.
- If the existing object is an **rvalue**, then it's a **move constructor**.

```
// Copy constructor, the parameter is an lvalue of the same type.
IntCell::IntCell( const IntCell &rhs ) {
    stored_ = new int{*rhs.value_};
}

// Move constructor, the parameter is an rvalue of the same type.
IntCell::IntCell( IntCell &&rhs ) : value_{rhs.value_} {
    rhs.value_ = nullptr;
}
```

When is either constructor called?

- During a declaration with initialization.

```
IntCell B = C;   // Copy constructor if C is lvalue; Move constructor is C
    is rvalue
IntCell B { C }; // same as above
```

- An object passed using call-by-value (rarely done).
- An object returned by value.

2.4.3 Copy assignment and move assignment (operator =)

The assignment operator is called when = is applied to two objects that have both been previously constructed. Given the expression `lhs = rhs`, then the state of `rhs` (right hand side) is copied into `lhs` (left hand side).

- If rhs is an **lvalue**, this is done using the copy assignment operator.
- If rhs is an **rvalue**, this is done using the move assignment operator.

```
// Copy assignment operator
IntCell & IntCell::operator=( const IntCell &rhs ) {
    if (this &= &rhs) {
        *value_ = *rhs.value_;
    }
    return *this;
}

// Move assignment operator
IntCell & IntCell::operator=( IntCell &&rhs ) {
    std::swap(value_, rhs.value_);
    return *this;
}
```

In C++11, the copy assignment operator could be implemented more idiomatically with a copy-and-swap idiom:

```
// Copy assignment operator
IntCell & IntCell::operator=( const IntCell &rhs ) {
    IntCell copy = rhs; // calls the copy constructor
    std::swap(*this, copy);
    return *this;
}
```

2.4.4 Defaults

It's often the case that the defaults "big five" are perfectly acceptable, so nothing need to be done. If a class consists of data members that are exclusively primitive types and objects for which the defaults make sense, the class defaults will usually make sense. Thus a class whose data members are `int`, `double`, `string`, and even `vector<string>` can accept defaults.

When do defaults fail? The defaults fail in a class that contains a data member that is a pointer:

- The default destructor does nothing to data members that are pointers. It's imperative that we `delete` them ourselves.
- The copy constructor and copy assignment operator both copy the value of the pointer (i.e., a memory address) rather than the object being pointed at. This means we end up with two class instances that contain pointers that point to the same objects being pointed at. This is known as **shallow copy**. However, we would typically expect a **deep copy**, in which a clone of the entire object is created.

> Thus, as a result, when a class contains pointers as data members, and deep semantics are important, we typically must implement the destructor, copy assignment, and copy constructors ourselves. Doing so removes the move defaults, so we also must implement move assignment and the move constructor. As a general rule, either you accept the default for all five operations, or you should declare all five, and explicitly define, default (use the keyword `default`), or disallow each (use the keyword `delete`). Generally we will defined all five.

NOTE: If you write any of the big-five, it would be good practice to explicitly consider all the others, as the defaults may be invalid or inappropriate. Changing a default method is an all or nothing situation.

When don't the defaults work? The most common situation in which the defaults do not work occurs when a data member is a pointer type and the pointer is allocated by some object member function (such as the constructor). A problem that might arisse is the default copy assignment and copy constructor doing shallow copies. Another problem is **memory leak**; an object allocated by a constructor might remain allocated and it's never reclaimed.

2.5 Templates

An algorithm is **type independent** if the logic of the algorithm does not depend on the type of items it handles. When we write C++ code for a type-independent algorithm or data structure, it's preferable to write the code once rather than recode it for each different type. This is accomplished by using **templates**.

2.5.1 Function templates

A **function template** isn't an actual function, but instead a pattern for what could become a function. For example, the following is a function template:

```
/*
Return the maximum item in array a.
Assumes a.size() > 0.
Comparable object must provide operator< and operator=.
*/

template <typename Comparable>
const Comparable & findMax( const vector<Comparable> &a ) {
    int maxIndex = 0;
    for (int i = 1; i < a.size(); i++) {
        if (a[maxIndex] < a[i]) {
            maxIndex = i;
        }
    }
    return a[maxIndex];
}
```

Using the function template would look as follows:

```
vector<int>    v1 = { 37, 12, 1, 89 };
vector<double> v2 = { 78.1, 89.8, 12.4, 1.1 };
vector<string> v3 = { 'hi', 'ha', 'ho', 'he' };
vector<IntCell> v4(75);

findMax(v1); // OK: Comparable = int
findMax(v2); // OK: Comparable = double
findMax(v3); // OK: Comparable = string
findMax(v4); // Illegal; IntCell doesn't implement operator< and thus not
   a Comparable
```

Thus, one of the limitations of function templates is that object passed to the function need to implement whatever operator the function uses internally. However, instead of relying on "hardcoded" conditions, a function that perform the required operation could be passed alongside the object. This type of functions are known as **function objects**.

2.6 Function objects

Previously we implemented `findMax` as a function template but it was only limited to objects that have an `operator<` function defined. Instead, we need to rewrite `findMax` to accept as parameters an array of object and a comparison function that explains how to decide which of two objects is the larger and which is the smaller. Instead of relying on the array objects knowing how to compare themselves, we completely decouple this information from the object in the arrays.

How do we pass a functions as parameters though? One way is to define a class with no data and one member function, and pass an instance of the class. In fact, the function is passed by placing it

inside an object, which is known as a **function object**.

The following is the simplest implementation of the function object idea:

```
#include <iostream>
#include <vector>
#include <string>

// Generic findMax, with a function object.
// Precondition: a.size() > 0.
// Comparator object is assumed to implement the isLessThan method.
template <typename Object, typename Comparator>
const Object & findMax( const std::vector<Object> &arr, Comparator cmp ) {
    int maxIndex = 0;

    for (int i = 1; i < arr.size(); i++) {
        if (cmp.isLessThan(arr[maxIndex], arr[i])) {
            maxIndex = i;
        }
    }

    return arr[maxIndex];
};

class StringComparisonByLength {
    public:
        bool isLessThan( const std::string &lhs, const std::string &rhs )
            {
            return lhs.length() < rhs.length();
        }
};

class StringComparisonCaseInsentitive {
    public:
        bool isLessThan( const std::string &lhs, const std::string &rhs )
            {
            return std::strcasecmp(lhs.c_str( ), rhs.c_str()) < 0;
        }
};

int main() {
    std::vector<std::string> greetings = {"hi", "hello", "bonjour", "HOLA"
        };

    std::cout << findMax( greetings, StringComparisonByLength{} ) << "\n";
    std::cout << findMax( greetings, StringComparisonCaseInsentitive{} )
        << "\n";

    return 0;
}
```

C++ function objects are implemented using this basic idea, but with some fancy syntax. First, instead of using a function with a name, we use operator overloading. Instead of the function being `isLessThan`, it is `operator()`. Second, when invoking `operator()`, `cmp.operator()(x,y)` can be shortened to `cmp(x,y)`.Third, we can provide a version of `findMax` that works without a function object that uses a default ordering; the implementation uses the Standard Library function object template less (defined in header file functional) to generate a function object that imposes the normal default ordering.

The following is the more idiomatic implementation:

```cpp
#include <iostream>
#include <vector>
#include <string>

// Generic findMax, with a function object.
// Precondition: a.size() > 0.
template <typename Object, typename Comparator>
const Object & findMax( const std::vector<Object> &arr, Comparator
    isLessThan ) {
    int maxIndex = 0;

    for (int i = 1; i < arr.size(); i++) {
        if (isLessThan(arr[maxIndex], arr[i])) {
            maxIndex = i;
        }
    }

    return arr[maxIndex];
};

// Generic findMax, using default ordering.
const Object &findMax( const vector<Object> &arr ) {
    return findMax(arr, less<Object>{} );
}

class StringComparisonByLength {
    public:
        bool operator()( const std::string &lhs, const std::string &rhs )
            {
            return lhs.length() < rhs.length();
        }
};

class StringComparisonCaseInsentitive {
    public:
        bool operator()( const std::string &lhs, const std::string &rhs )
            {
            return std::strcasecmp(lhs.c_str( ), rhs.c_str()) < 0;
        }
};

int main() {
    std::vector<std::string> greetings = {"hi", "hello", "bonjour", "HOLA"
        };

    std::cout << findMax( greetings, StringComparisonByLength{} ) << "\n";
    std::cout << findMax( greetings, StringComparisonCaseInsentitive{} )
        << "\n";
    std::cout << findMax( greetings ) << "\n";

    return 0;
}
```

2.6.1 Class templates

In its most simplest form, a class template works much like a function template. The following `MultiCell` is an implementation that is like `IntCell`, but works for any type `Object`, provided that `Object` has a zero-parameter constructor, a copy constructor, and a copy assignment operator.

```
template <typename  Object>
class MemoryCell {
    public:
        explicit MemoryCell( const Object &initialValue = Object{} ) :
           _value{initialValue} {}
        const Object & read() const { return _value; }
        void write( const Object &x ) { _value = x; }

    private:
        Object _value;
};
```

Notice that

- `Object` is passed by constant reference.
- the default parameter for the constructor is not 0, because 0 might not be a valid Object. Instead, the default parameter is the result of constructing an Object with its zero-parameter constructor.

> If we implement class templates as a single unit, then there is very little syntax baggage. Many class templates are, in fact, implemented this way because, currently, separate compilation of templates does not work well on many platforms. **Therefore, in many cases, the entire class, with its implementation, must be placed in a .h file**. Popular implementations of the STL follow this strategy.

2.7 Using matrices

This will be implemented by using a vector of vectors (e.g., a vector of **`int`** vectors).

```
#ifndef MATRIX_H
#define MATRIX_H

#include <vector>

template <typename Object>
class Matrix {
    public:
        /*
         * Create _array as having rows entries each of type vector<Object
            >.
         * We have a rows zero-length vectors of Object so each row is
            resized
         * to have cols columns. Thus this creates a two-dimensional array
            .
         */
        Matrix( int rows, int cols ) : _array(rows) {
            for (auto &row : array) {
                row.resize(cols);
            }
        }

        Matrix( std::vector<std::vector<Object>> v ) : _array{ v } { }

        Matrix( std::vector<std::vector<Object>> &&v ) : _array{ std::move
            (v) } { }

        /*
         * Array indexing operator. This returns the row (a vector<Object>
            at
         * the index row.
         */
        const std::vector<Object> & operator[]( int row ) const {
            return _array[row]
        }

        int numrows() const {
            return _array.size();
        }

        int numcols() const {
            return numrows() ? _array[0].size() : 0;
        }

    private:
        /*
        A matrix is represented by _array, a vector of vector<Object>.
        */
        std::vector<std::vector<Object>> _array;
};

#endif
```

2.8 Class problems

1. What's $1 + 2 + 3 + ... + n$? Prove it.

$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$

Proof by induction:

- **Base case:** When $n = 1$, then $1 = 1(1 + 1)/2 = 1$.
- **Inductive step:** Since $1 = (1 * 2)/2$, the statement $P(1)$ is true. Assume that $P(k)$ is true for an arbitrary positive integer k. We show that $P(k + 1)$ is true. In other words,

$$1 + 2 + 3 + ... + (k + 1) = \frac{(k + 1)(k + 2)}{2}$$

Thus

$$\begin{aligned} 1 + 2 + 3 + \cdots + (k + 1) &= (1 + 2 + 3 + \cdots + k) + (k + 1) \\ &= \frac{k(k + 1)}{2} + (k + 1) \\ &= \frac{k(k + 1)}{2} + (k + 1) \\ &= \frac{(k + 1)(k + 2)}{2} \end{aligned}$$

Therefore, by the principle of mathematical induction, $P(n)$ is true for every positive integer n.

2. What's $1 + 2 + 4 + \cdots + 2^n$? Prove it.

$$\begin{aligned} &n = 0 => 1 => 1 \\ &n = 1 => 1 + 2 => 3 \\ &n = 2 => 1 + 2 + 4 => 7 \\ &n = 3 => 1 + 2 + 4 + 8 => 15 \\ &... \\ &n = n => 1 + 2 + \cdots + 2^n => 2^{(n+1)} - 1 \end{aligned}$$

Thus, $1 + 2 + 4 + \cdots + 2^n = 2^{(n+1)} - 1$.

Proof by induction:

- **Base case:** When $n = 0$, then $1 = 2^{(0+1)} - 1 = 1$. When $n = 1$, then $2^0 + 2^1 = 3 = 2^{(1+1)} - 1 = 3$.
- **Inductive step:** Since $1 = 2^{(0+1)} - 1$, the statement $P(1)$ is true. Assume that $P(k)$ is true for an arbitrary positive integer k. We show that $P(k + 1)$ is true. In other words, we show that

$$1 + 2 + 4 + \cdots + 2^{(k+1)} = 2^{(k+2)} - 1.$$

Thus,

$$\begin{aligned} 1 + 2 + 4 + \cdots + 2^{(k+1)} &= (1 + 2 + 4 + \cdots + 2^k) + 2^{(k+1)} \\ &= 2^{(k+1)} - 1 + 2^{(k+1)} \\ &= 2^{(k+1)} + 2^{(k+1)} - 1 \\ &= 2 \cdot 2^{(k+1)} - 1 \\ &= 2^{(k+2)} - 1 \end{aligned}$$

Therefore, by the principle of mathematical induction, $P(n)$ is true for every non-negative integer n.

3. If $A_0 = 1$ and $A_n = 2A_{(n-1)} + 1$, what's a closed formula for A_n? Prove it.

$$\begin{aligned} A_1 &= 2 \cdot A_0 + 1 = 3 \\ A_2 &= 2 \cdot A_1 + 1 = 7 \\ A_3 &= 2 \cdot A_2 + 1 = 15 \\ A_4 &= 2 \cdot A_3 + 1 = 31 \\ &\cdots \\ A_n &= 2 \cdot A_{(n-1)} + 1 = 2^{(n+1)} - 1 \end{aligned}$$

3 Lecture 3: Algorithm analysis

An **algorithm** is a clearly defined set of simple instructions which must are to be followed in order to solve a particular problem.

3.1 Mathematical background

Algorithm analysis is grounded on mathematics and the definitions below set up a formal framework to study algorithms. These definitions try to establish a **relative order among functions**. Given two functions, there are usually points where one function is smaller than the other so it doesn't make sense to claim, for instance, $f(N) < g(N)$. Instead, we compare their **relative rates of growth**.

Big-Oh $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq c \cdot f(N)$ when $N \geq n_0$.

Informally, the growth rate $T(N)$ is less than or equal to that $f(N)$.

Big-Omega $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq c \cdot g(N)$ when $N \geq n_0$.

Informally, the growth rate $T(N)$ is greather than or equal to that $g(N)$.

Big-Theta $T(N) = \Theta(h(N))$ **if and only if** $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

Informally, the growth rate $T(N)$ equals the growth rate of $h(N)$. This means that $T(N)$ is eventually *sandwiched* between two different constant multiples of $h(N)$.

Little-Oh $T(N) = o(p(N))$ if, for all positive constants c, there exist an n_0 such that $T(N) < c \cdot p(N)$ when $N > n_0$.

Informally, the growth rate $T(N)$ is less than the growth rate of $f(N)$. This is different from Big-Oh, given that Bog-Oh allows the possibility that the growth rates are the same.

NOTE: In the above definitions, both c and n_0 are constants and they cannot depend on N. If you see yourself saying "take $n_0 = N$" or "take $c = log_2 N$" in an alleged Big-Oh proof, then you need to start with choices of c and n_0 that are indepedent of N.

3.2 Notation

3.2.1 Big-Oh notation

When we say that $T(N) = O(f(N))$, we're guaranteeing that the function $T(N)$ grows at a rate no faster than $f(N)$; thus $f(N)$ is an **upper bound** on $T(N)$. Alternatively, we could say that $T(N)$ is

bounded above by a constant multiple of $f(N)$.

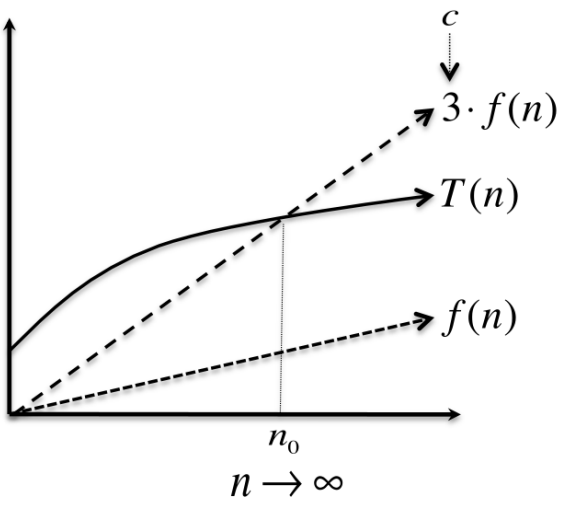


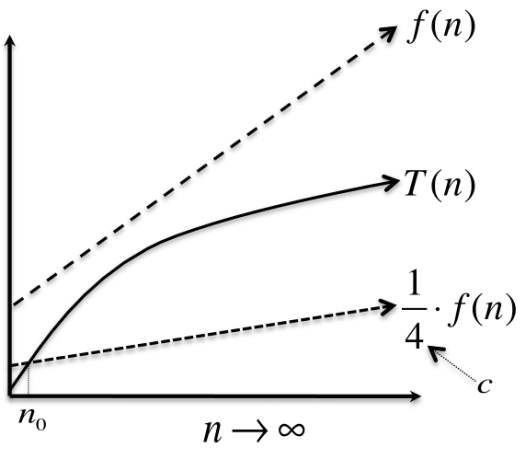
Figure 2.1: A picture illustrating when $T(n) = O(f(n))$. The constant c quantifies the "constant multiple" of $f(n)$, and the constant n_0 quantifies "eventually."

Figure 1: Big-Oh

For example, $T(N) = O(N^2)$ means that the function $T(N)$ grows at a rate no faster than N^2; its growth could equal that of N^2 but it could never surpass it.

3.2.2 Big-Omega notation

When we say that $T(N) = \Omega(g(N))$, we're guaranteeing that the function $T(N)$ grows at a rate no lower than $g(N)$; thus $g(N)$ is a **lower bound** on $T(N)$. Alternatively, we could say that $T(N)$ is **bounded below** by constant mutiple of $g(N)$.



$T(n)$ again corresponds to the function with the solid line. The function $f(n)$ is the upper dashed line. This function does not bound $T(n)$ from below, but if we multiply it by the constant $c = \frac{1}{4}$, the result (the lower dashed line) does bound $T(n)$ from below for all n past the crossover point at n_0. Thus $T(n) = \Omega(f(n))$.

Figure 2: Big-Omega

For example, $T(N) = \Omega(N^2)$ means that the function $T(N)$ grows at a rate no lower than N^2; its growth could equal that of N^2 but it could never drop below it.

Whenever talking about Big-Oh, there's always an implication about Big-Omega. For instance, $T(N) = O(f(N))$ (i.e., $f(N)$ is an **upper bound** on $T(N)$) implies that $f(N) = \Omega(T(N))$ (i.e., $T(N)$ is a **lower bound** on $f(N)$). As an example, N^3 grows faster than N^2, so we can say that $N^2 = O(N^3)$ or $N^3 = \Omega(N^2)$.

3.2.3 Big-Theta notation

When we say that $T(N) = \Theta(h(N))$, we're guaranteeing that the function $T(N)$ grows at the same rate as $g(N)$. However, when two functions grow at the same rate, then the decision of whether or not to signify this with $\Theta()$ can depend on the context.

3.3 Typical growth rates

Function	Name
c	Constant
$log_2 N$	Logarithmic
$log^2 N$	Log-squared
N	Linear
$NlogN$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

3.4 Rules

Rule 1 If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

1. $T_1(N) + T_2(N) = O(f(N) + g(N))$. Less formally it is $O(\max(f(N), g(N)))$, and
2. $T_1(N) \cdot T_2(N) = O(f(N) \cdot g(N))$

Rule 2 If $T(N)$ is a polynomial of degree k, then $T(N) = \Theta(N^k)$.

Rule 3 $log^k N = O(N)$ for any constant k. In other words, logarithms run at a lower rate than linear functions which means logarithms grow very slowly.

3.5 Few points

- In simple terms, the goal of asymptotic notation is *to suppress constants factors and lower-order*. Thus, they aren't included in Big-Oh. For instance, don't say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$. In both cases, the correct form is $T(N) = O(N^2)$.

When analyzing the running time of an algorithm, why would we want to throw away information like constant factors and lower-order terms? 1) Lower-order terms become increasingly irrelevant as you focus in large inputs, which are the inputs that require algorithmic ingenuity and 2) constant factors are generally highly dependent on the details of the environment (e.g., programming language, architecture, compiler, etc,) and thus ditching them allows to generalize by not commiting ourselves to a specific programming language, architecture, etc.

- The relative growth rates of two functions $f(N)$ and $g(N)$ can always be determined by computing $lim_{N\to\infty} \frac{f(N)}{g(N)}$, using L'Hopital's rule if necessary. The limit can have four possible values:

  - The limit is 0, meaning that $f(N) = O(g(N))$.
  - The limit is $c \neq 0$, meaning that $f(N) = \Theta(g(N))$.
  - The limit if ∞, meaning that $g(N) = o(f(N))$.
  - The limit doesn't exist and thus no relation exist.

- Stylistically it's bad to say that $f(N) \leq O(g(N))$ since the inequality is already implied by the definition of Big-Oh.
- It's wrong to write $f(N) \geq O(g(N))$, because it doesn't make sense.

3.6 Model of computation

The model of computation is basically a normal computer with the following characteristics:

- Instructions are executed sequentially.
- The model has the standard repertoires of simple instructions, such as addition, multiplication, comparison, and assignment. Unlike real computers, this computer takes exactly one time unit to do anything.
- The computer has fixed-size (e.g., 32-bit) integers and no fancy operations (e.g., matrix invertion, sorting, etc.).
- The computer has infinite memory.

3.7 What to analyze

The most important resource to analyze is generally the *running time* and typically, the size of the input is the main consideration. We define two functions, $T_{avg}(N)$ (for the average-case running time) and $T_{worst}(N)$ (for the worst-case running time) used by an algorithm on input of size N. It's worth noting that $T_{avg}(N) \leq T_{worst}(N)$ (i.e., $T_{avg}(N)$ has a lower rate growth than that of $T_{worst}(N)$).

3.7.1 Types of performance

Best-case performance Although it can be ocassionally analyzed, it's often of little interest since it doesn't represent typical behavior.

Average-case performance It often reflects typical behavior, however it doesn't provide a bound for all input and it can be difficult to compute. Furthermore, it's also hard to define what's the average input

Worst-case performance It represents a guarantee for performance on any possible input. This is the quantity generally required because it provides a bound for all input.

3.8 Running-time calculations

When computing a Big-Oh running time, there are several general rules:

For loops The running time of a `for` loop is *at most* the running time of the statements inside the `for` loop times the number of iterations.

Nested loops Analyze these inside out. The total running tome of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

For example, the following program is $O(N^2)$ because the inner loop does N iterations and the outer loops does N too. This amounts to $N \times N = N^2$ iterations. The first assignment counts as one operation (thus $O(N)$) while the innermost statement counts for 2 operations (1 multiplication and 1 assignment). Thus, to be more precise the program's running time is $1 + 2N^2$, however Big-Oh suppresses constant factors and lower-order terms.

```
k ← 0
for [1, n] → i:
    for [1, n] → j:
        k ← i * j
```

Consecutive statements They just add (which means that the maximum is the one that counts).

For example, the following program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is ultimately $O(N^2)$ which dominates $O(N)$:

```
for [0, n) → i:
    a[i] = 0

for [0, n) → i:
    for [0, n) → j:
        a[i] += a[j] + i + j
```

If/else For the fragment `if condition { S1 } else { S2 }`, the running time of an `if`/`else` statement is never more than the running time of the test plus the larger of the running times of `S1` and `S2`.

3.9 Sample problems

3.9.1 Sum of cubes

```
Input: a positive integer n.
Output: the sum of all cubes from 1 to n^3.

SumOfCubes(n):
    sum ← 0                 # O(1)
    for [1, n] → i:         # O(n)
        sum += i * i * i    # O(4), 1 assignment, 1 addition and 2 products
    return sum              # O(1)
```

Thus, for `SumOfCubes(N)= 1 + 4n + 1 = 4n + 2 = O(n)`. We discard both the constant factors and the lower-order terms.

3.9.2 Factorial

The factorial is defined as

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Implemented recursively, the algorithm is as follows:

```
Input: a non-negative integer n
Output: the factorial of n

Factorial(n):
    if n ≤ 1:
        return 1
    else:
        return n * Factorial(n-1)
```

However, this is a thinly veiled **for** loop. In this case, the analysis involves the recurrence relation $T(n) = 1 + T(n-1)$ for $n > 1, T(1) = 2$ that needs to be solved:

$$
\begin{aligned}
T(N) &= 1 + T(n-1) \\
&= 1 + (1 + T(-2)) \\
&= 1 + (1 + (1 + T(n-3))) \\
&= ... \\
&= 1 + (1 + (1 + T(n-k))) \; k \text{ 1's}
\end{aligned}
$$

$$
\begin{aligned}
T(N) &= k + T(n-k) \\
&= (n-1) + T(n-(n-1)) \\
&= (n-1) + T(1) = (n-1) + 2 = n + 1
\end{aligned}
$$

Thus, $T(n) = O(n)$.

3.9.3 Maximum subsequence sum problem

> Given (possibly negative) integers $A_1, A_2, ..., A_N$, find the maximum value of $\sum_{k=1}^{i} A_k$.

In other words, given a one-dimensional array of numbers, find a contiguous subarray with the largest sum. For example, with the input $-2, 11, -4, 13, -5, -2$ the answer is $11 - 4 + 13 = 20$.

This problem is interesting mainly because there are many algorithms to solve it, and the performance of these algorithms varies drastically.

	Algorithm Time			
Input Size	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

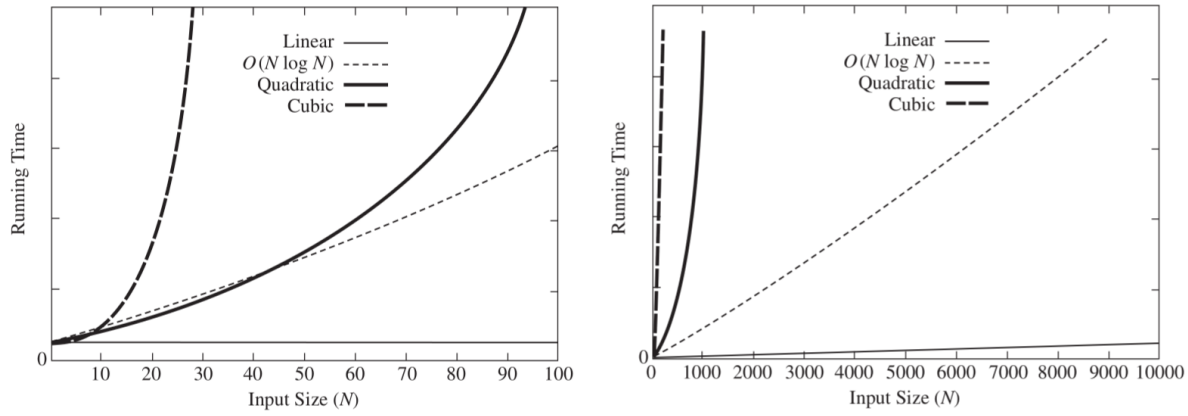


Figure 3: Plot (N vs. time) of various algorithms

For algorithm 4 (linear), as the problem size increases by a factor of 10, so does the running time. For algorithm 3 (quadratic), a tenfold increase in input size yields roughly hundrefold (10^2) increase in running time.

3.9.4 Algorithm 1 (brute force)

`implementations/max-subsequence-sum`:

```
Input: array A of integers.
Output: the maximum positive subsequence sum.

MaxSubsequenceSum(A):
    maxSum ← 0
    for [0, A.size) → i:
        for [i, A.size) → j:
            currentSum ← 0
            for [i, j] → k:
                currentSum += A[k]
            if currentSum > maxSum:
                maxSum = currentSum
    return maxSum
```

For this algorithm, there's N starting places, average $\frac{N}{2}$ lengths to check, and average $\frac{N}{4}$ numbers to add. Thus we have $O(N^3)$.

3.9.5 Algorithm 2 (brute force)

implementations/max-subsequence-sum:

```
Input: array A of integers.
Output: the maximum positive subsequence sum.

MaxSubsequenceSum( A ):
    maxSum ← 0
    for [0, A.size) → i:
        currentSum ← 0
        for [i, A.size) → j:
            currentSum += A[j]
            if currentSum > maxSum:
                maxSum = currentSum
    return maxSum
```

For this algorithm, the innermost **for** loop has been removed. There's N starting places and an average $\frac{N}{4}$ numbers to add. Thus we have $O(N^2)$.

3.9.6 Algorithm 3

implementations/max-subsequence-sum:

```
Input: array A of integers.
Output: find the maximum sum in subarray spanning A[LEFT..RIGHT]. It does
not attempt to mantain actual best sequence.

MaxSumRec( A, LEFT, RIGHT ):
    # Base case
    if LEFT = RIGHT:
        if A[LEFT] > 0:
            return A[LEFT]
        else
            return 0

    center ← (LEFT + RIGHT) div 2
    maxLeftSum ← MaxSumRec(A, LEFT, center)
    maxRightSum ← MaxSumRec(A, center + 1, RIGHT)

    maxLeftBorderSum ← 0
    leftBorderSum ← 0

    for [center, LEFT] → i:
        leftBorderSum += A[i]
        if leftBorderSum > maxLeftBorderSum:
            maxLeftBorderSum ← leftBorderSum

    maxRightBorderSum ← 0
    rightBorderSum ← 0

    for [center + 1, RIGHT] → j:
        rightBorderSum += A[j]
        if rightBorderSum > maxRightBorderSum:
            maxRightBorderSum ← RightBorderSum

    return max(maxLeftSum, maxRightSum, maxLeftBorderSum +
       maxRightBorderSum)

# Driver for divide-and-conquer maximum contiguous subsequence sum
   algorithm.
MaxSubsequenceSum( A ):
    return MaxSumRec(A, 0, A.size - 1)
```

3.9.7 Algorithm 4

implementations/max-subsequence-sum:

```
Input: array A of integers.
Output: the maximum positive subsequence sum.

MaxSubsequenceSum( A ):
    maxSum ← 0
    currentSum ← 0

    for [0, A.size] → j:
        currentSum += A[j]

        if currentSum > maxSum:
            maxSum ← currentSum
        else if currentSum < 0:
            currentSum = 0

    return maxSum
```

4 Lecture 4: Algorithm analysis (cont.) and lists/stacks/queues

4.1 Binary search

> Given an integer X and integers $A_0, A_1, \ldots, A_{N-1}$, which are **presorted** already in memory, find index i such that $A_i = X$, or return -1 if X is not in the group of integers.

`implementations/binary-search`

```
Input: array A of elements and item x which we are searching for.
Output: item where item is found or -1 if not found.

BinarySearch( A, x ):
    low ← 0
    high ← A.size - 1
    while low ≤ high:
        mid ← (low + high) / 2
        if A[mid] < x:
            low = mid + 1
        else if A[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1
```

4.1.1 Running time

We have the following recurrence relation $T(N) = 1 + T(N/2)$ and $T(1) = 1$. We must solve it to find out the algorithm's running time:

$$
\begin{aligned}
T(N) &= 1 + T(N/2) \\
&= 1 + 1 + T(N/2^2) \\
&= 1 + 1 + 1 + T(N/2^3) \\
&\dots \\
&= k + T(N/2^k)
\end{aligned}
$$

If N is a power of 2 (i.e., $N = 2^k$ with $k = log(N)$) we'll have:

$$
\begin{aligned}
T(N) &= k + T(N/2^k) \\
&= k + T(1) = log(N) + 1
\end{aligned}
$$

This means that $T(N) = O(logN)$.

4.2 List/stacks/queue ADTs

An **abstract data type** (ADT) is a set of objects (lists, sets, graphs, etc.) together with a set of related operations (add, remove, size, etc.). They provide a template for the objects, not how the objects and their respective set of operations are implemented.

4.2.1 The List ADT

We usually deal with a general list of the form $A_0, A_1, A_2, \dots, A_{N-1}$. We say that the size of this list is N. We will call the special list of size 0 an **empty list**. The **position** of the element A_i in a list is i.

The List ADT could describe the followin operations:

find(x) Return the position of the occurrence of item x.
insert(x, i) Insert some element x at position i.
remove(i) Remove some element at position i.

4.2.2 Vector in the STL

Pros:

- Constant tome indexing
- Fast to add data at the end (not the front).

Cons:

- Slow to add data in the middle.
- Inefficient for searches.

Operations

- `void push_back(const T &x)` - add `x` at the end of the list.
- `void pop_back()` - remove element at the end of the list.
- `const T &back()const` - return the element at the end of the list.
- `const T &front()const` - return the element at the front of the list.
- `void push_front(const T &x)` - add `x` to the front of the list.
- `void pop_front()` - remove the element at the front of the list.

4.2.3 List in the STL

Pros:

- Implemented as a doubly linked list.
- Fast insertion/removal of items in any position.

Cons:

- No indexing.
- Inefficient for searches.

Operations

- `T & operator[](int idx)` - return element at index `idx` with no bounds checking.
- `T &at(int idx)` - return element at index `idx` with bounds checking.
- `int capacity()conts` - return internal capacity of vector‘.
- `void reserve(int new_capacity)` - set new capacity and possibly void expansion of vector.

4.3 Containers

A **container** is a holder object that stores a collection of other objects. This is usually implemented as a class templates which provides it with great flexibility for the types supported as elements.

A container

- manages the storage space for its elements and provides member functions to access them either directly or through **iterators**.
- replicates structures commonly used in programming such as dynamic arrays (e.g., vectors), queues, stacks, heaps (e.g., priority queues), linked lists, trees (e.g., sets), associative arrays (e.g., maps), etc.

4.4 Iterators

Some operations on lists require the notion of a position. In the STL, a position is represented by some nested type known as an **iterator**. In particular, for a `list<string>`, the position is represented by the type `list<string>::iterator`; for a `vector<int>`, `Vector<int>::iterator`.

4.4.1 How to get an iterator

The STL lists (and all other STL containers) define a pair of methods:

- `iterator begin()`: returns an appropriate iterator representing the first item in the container.
- `iterator end()`: returns an appropriate iterator representing the endmarker in the container. This endmarker is "out-of-bounds".

4.4.2 Iterator methods

- `itr++` and `++itr`: advances the iterator `itr` to the next location.
- `*itr`: returns a reference, which may or may not be modifiable, to the object stored at iterator `itr`'s location.
- `itr1 == itr2`: returns true if iterators `itr1` and `itr2` refer to the same location and false otherwise.
- `itr1 != itr2`: returns true if iterators `itr1` and `itr2` refer to a different location and false otherwise.

For example the code:

```
for (int i = 0; i <!= v.size(); i++) {
    std::cout << v[i] << "\n";
}
```

could be rewritten as follows using iterators:

```
for (vector<int>::iterator itr = v.begin(); itr != v.end(); itr++) {
    std::cout << *itr << "\n";
}
```

Alternatively:

```
vector<int>::iterator itr = v.begin();
while (itr != v.end()) {
    std::cout << *itr++ << "\n";
}
```

4.4.3 Container operations that require iterators

- `iterator insert( iterator pos, const T & x )`: adds `x` into the list, prior to the position by the iterator `pos`. This is a constant operator for `list`, but not for `vector`. The return value is an iterator representing the position of the inserted item.
- `iterator erase( iterator pos )`: removes the object at position and return the position of the element that followed `pos` prior to the call. It invalidates `pos`, making it stale.
- `iterator erase( iterator start, iterator pos )`: removes all items beginning at position up to, but not including `end`. An entire list `c` can be erased by `c.erase( c.begin(), c.end())`.

4.4.4 Example: Using `erase` on a list

```
/*
 * @brief Deletes every other element from lst, starting from the first
    item.
 * @param lst a list, or any object that supports iterators and erase.
 */
template <typename Container>
void removeEveryOtherItem( Container &lst ) {
    typename Container::iterator itr = lst.begin(); // could be shortened
       to
                                                    // auto itr = lst.
                                                       begin();
    while (itr != lst.end()) {
        itr = lst.erase(itr);
        if (itr != lst.end()) itr++;
    }
}
```

4.4.5 `const_iterators`

The result of `*itr` is both the value of the item that the iterator is viewing but also the item itself. This is very powerful but also introduces some complications.

Let's analyze the following routine that works for both `vector` and `list` and runs in linear time:

```
template <typename Container, typename Object>
void change( Container &c, const Object &newValue ) {
    typename Container::iterator itr = c.begin();
    while (itr != c.end()) {
        *itr = newValue; // set current iterator's value to newValue
        itr++;           // advance the pointer to next item
    }
}
```

The potential problem arises if `Container c` was passed to a routine using call-by-constant reference, meaning we would expect that not changes would be allowed to `c`, and the compiler would ensure this by nota allowing calls to any `c`'s mutators. For example, consider the following code that prints a `list` of integers but also tries to do some changes:

```
void print( const list<int> &lst, ostream &out = cout ) {
    typename Container::iterator itr = lst.begin();
    while (itr != lst.end()) {
        out << *itr << "\n";
        *itr = 0;   // <= This is the suspect.
        itr++;
    }
}
```

The solution provided by the STL is that every collection contains not only an `iterator` nested type but also `const_iterator` nested type. The main difference between them is that `operator*` for `const_iterator` returns a constant reference, and thus `itr*` for a `const_iterator` cannot appear on the left-hand side of an assignment statement.

Further the compiler will force you to use a `const_iterator` to traverse a *constant* collection and does so by providing two versions versions of `begin` and two versions of `end`:

- `iterator begin()`
- `const_iterator begin()const`
- `iterator end)`
- `const_iterator end()const`

Note: The two versions of `begin` can be in the same class only because of the const-ness of a method (whether an accessor or mutator) is considered to be part of the signature. The trick is overloading `operator[]`.

Using `auto` to declare your iterators means the compiler will deduce for you whether an `iterator` or `const_iterator` is substituted. This exempt the programmer from having to keep track of the correct iterator type and is precisely one of the intended uses of `auto`.

4.4.6 Printing a container

```
/*
 * @brief Prints out the container on the output stream.
 * @param c a given container.
 * @param out an output stream.
 */
template <typename Container>
void print( const Container &c, ostream &out = cout ) {
    if (c.empty()) {
        out << "(empty)";
    }
    else {
        typename Container::iterator itr = begin(c);
        out << "[" << *itr++; // print first item
        while (itr != end(c)) {
            out << ", " << *itr++;
        }
        out << "]" << "\n";
    }
}
```

5 Lecture 5: Lists, Stacks, Queues, and the STL

5.1 Implementation of a vector

The main details of the implementation of the `Vector` class is as follows:

- It'll maintain the primitive array (via a pointer variable to the block of allocated memory), the array capacity, the current number of stored items.
- It'll implement the "big-five" to provide **deep-copy** semantincs.
- It'll provide the `resize` and `reserve` methods that will change the size of the `Vector` and the capacity of the `Vector` respectively.
- It'll overload the `operator[]`.
- It'll provide several basic methods, such as `size`, `empty`, `clear`, `back`, `pop_back`, and `push_back`.
- It'll provide support for the nested types `iterator` and `const_iterator`, and associated `begin` and `end` methods.

Implementation: implementations/Vector

Why create a class `Vector`? Is it better than a simple array? There are few reasons why creating `Vector` class template makes sense. The main one is that `Vector` will be a first-clas type, which means that 1) it can be copied, and 2) the memory it uses can be reclaimed back (via its destructor). This is not the case for primitie array in C++.

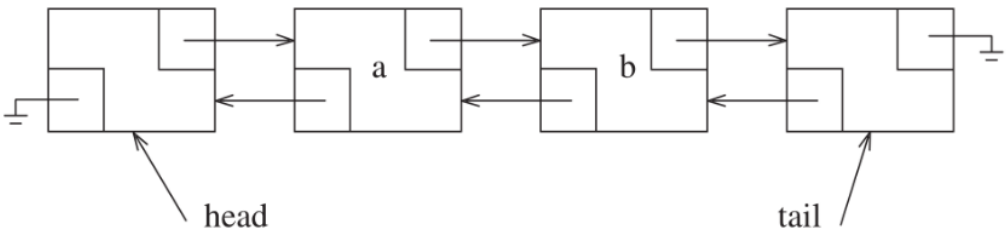
5.2 Implementation of a list

The main details of the implementation of the `List` class template is as follows:

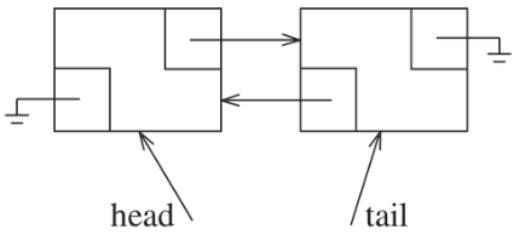
- The underlying data structure is a **doubly linked list** and thus we must mantain pointers to both ends of the list. This is in order to mantain constant time cost per operation, so as long as the operation occurs at a known position. The known position can be either end or at a position specified by an iterator.
- Four classes will be provided to better compose the `List` class template:
  - The `List` class itself, which contains links to both ends, the size of the list, and several methods.
  - The `Node` class, which is likely to be a private nested class. A node contains the data and pointers to the previous and next nodes, along with appropriate constructors.

- The `const_iterator` class, which abstracts the notion of a position, and is a public nested class. It stores a pointer to the curren node, a provodes implementation of the basic iterator operations.
- The `iterator` class, which abstracts the notion of a position, and is a public nested class. It has similar functionality to `const_iterator`, except that `operator*` returns a **reference** to the item being viewed, rather than a **constant reference**.
- Extra nodes are at both the front and end of the list. The node at the front represents the beginning marker while the one at the end represents the endmarker. These are the **sentinel nodes**; the one at the front is the **header node** and the one at the end is the **tail node**. Using the sentinels simplify the coding by removing a host of special cases (e.g., removing the first node).

Implementation: implementations/List



A doubly linked list with header and tail nodes

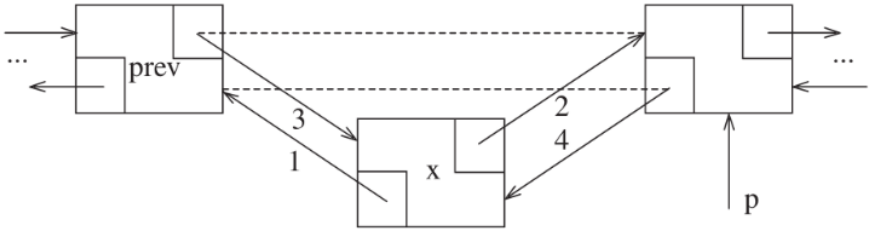


An empty doubly linked list with header and tail nodes

Figure 4: Doubly linked list

5.2.1 Insertion

```
Node *newNode = new Node{ x, p->prev, p };  // Steps 1 and 2
p->prev->next = newNode;                    // Step 3
p->prev = newNode;                          // Step 4
```



Insertion in a doubly linked list by getting a new node and then changing pointers in the order indicated

Figure 5: Adding a node

5.2.2 Erasure

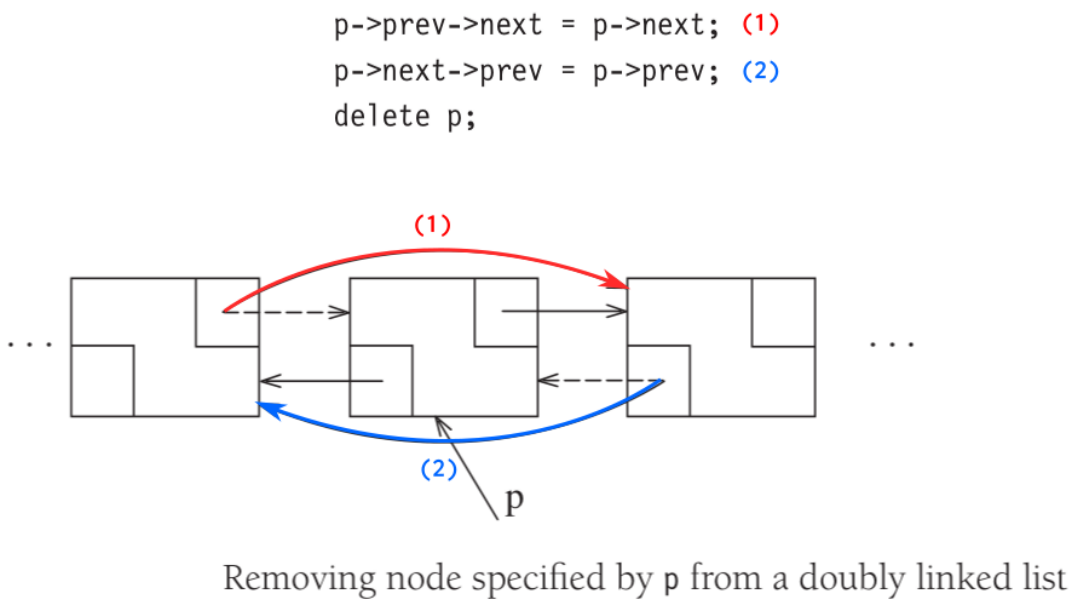


Figure 6: Removing a node

5.3 The Stack ADT

A **stack** (sometimes known as a LIFO (*last in, first out*) list) is a special list where insertions and deletions can be performed only from the end of the list (a.k.a., the **top**).

The fundamental operations on a stack are:

push Push an item to the top of the stack. Equivalent to `insert`.
pop Pop the most recently inserted item. The topmost item can be examined by using the `top` routine.

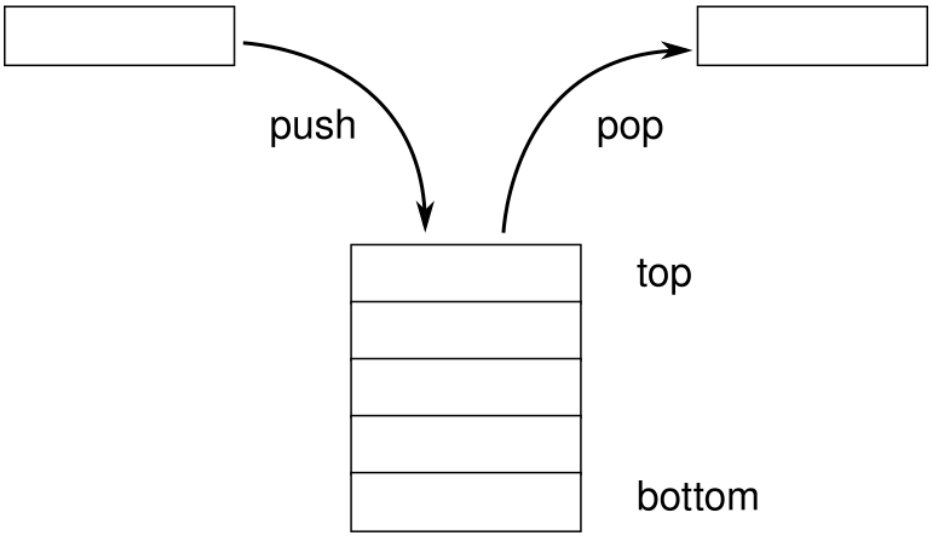


Figure 7: Stack model

5.3.1 Implementations

Two population implementations of stacks are:

- **Linked list implementation**

This implementation of a stack uses a singly linked list. A `push` operation is performed by inserting at the front of the list, a `pop` operation is performed by deleting the element at the front of the list, and a `top` operation merely examines the element at the front of the list by returning its value.

Implementation: implementations/LLStack

- **Array implementation**

This implementation of a stack avoids pointers and is usually the most popular solution. It uses the `back`, `push_back`, and `pop_back` implementation from `vector`. Associated with each stack are `array` to stores the items and `top_of_stack` which is -1 for an empty stack.

Implementation: implementations/ArrayStack

5.3.2 Applications

After the array, the stack is probably the most fundamental data structure in computer science and for this reason, the application of stacks can be found in many places. Some of them are:

- **Balancing symbols**

Given that compiler check programs for syntax error, a useful tool to check for the lack of symbols that might cause errors is a program that checks whether everything is balanced (i.e., every right brace, bracket, and parenthesis must correspond to its left counterpart). For instance, the sequence `[()]` is legal, but `[(])` is wrong. This algorithm uses a stack and could be described as follows:

> Make an empty stack. Read characters until end of file. If the character is an opening symbol, push it onto the stack. If it is a closing symbol and the stack is empty, report an error. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error. At end of file, if the stack is not empty, report an error.

- **Postfix expressions**

This notation, also known as **reverse Polish notation**, is a mathematical notation in which operators follow their operands, in contrast to Polish notation, in which operators precede their operands. For example, the infix expression $((15 \div (7 - (1 + 1))) \times 3) - (2 + (1 + 1))$ can be written in reverse Polish notation as $15\,7\,1\,1\,+\,-\,\div\,3\,\times\,2\,1\,1\,+\,+\,-$ which yields 5 regardless of the direction of the evaluation. An algorithm that uses a stack and that processes the expression from left to right is described below:

```
Input: EXPR is an expression in reverse Polish notation.
Output: EXPR

EvaluatePostfixExpression( EXPR ):
    stack ← []
    for EXPR → token:
      if token = operator:
          operand_2 ← stack.pop()
          operand_1 ← stack.pop()
          result ← evaluate operator_1 and operand_2 with operator
          stack.push(result)
      else if token = operand:
          stack.push(token)
    return stack.pop()
```

5.4 The Queue ADT

A **queue** (sometimes known as a FIFO (*first in, first out*) list) is a speciallist where insertions are performed at one end and deletions are performed at the other end.

The fundamental operations on a queue are:

enqueue Insert an element at the end of the list (called the rear).
dequeue Delete (and return) the element at the start of the list (called the front).

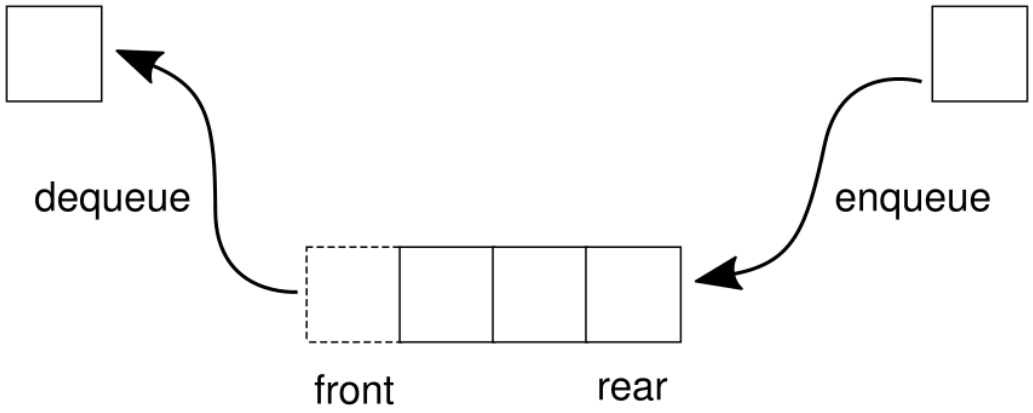


Figure 8: Queue model

5.4.1 Implementations

Two population implementations of stacks are:

- **Linked list implementation**.

This is implementation is similar to its counterpart stack implementation.

Implementation: implementations/LLQueue

- **Array implementation**

For this implementation, we keep an array, `array`, and the positions `front` and `back`, which represents the ends of the queue. Likewise, we keep track of the number of items currently in the queue, `current_size`. To `enqueue` an element `x`, we increment `current_size` and `back`, then

set `_array[back] = x`. To `dequeue` an element, we set the return value to `array[front]`, decrement `current_size`, and increment `front`.

However there's one potential problem with this implementation. After a certain number of operations, a queue might appear full since `back` is now at the last array index, and the next `enqueue` would be in a nonexistent position. A simple solution would be to wrap around to the beginning whenever `front` or `back` gets to the end of the array. This is known as a **circular array** implementation for which modular arithmetic is used.

Implementation: implementations/ArrayQueue

5.4.2 Applications

There are many algorithms that used queues to give efficient running times. Some examples are:

- Jobs submitted to a printer are arranged in order of arrival. Thus, they're essentially placed in a queue.
- Virtually every real-life line is supposed to be a queue. For instance, lines at ticket counters are queues, because the service is first-come first-served.
- There are many network setups of personal computers in which the disk is attached to one machine, known as the file server. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.

5.5 Trees

Recursively, a **tree** is either empty or consists a root node r and zero or more non-empty subtrees whose roots are connected by an edge.

- Used to implement file systems of several popular OSs
- Used to evaluate arithmetic expressions
- Support O(logN) search
- Set and map classes.

6 Lecture 6: Trees

6.1 Preliminaries

A **tree** is a widely used abstract data type (ADT) that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Trees (more specifically binary trees) are really important so a certain terminoly has developed for them:

- A **node** is a structure which may contain a value or condition, or represent a separate data structure.
- An **edge** is connection between one node and another.
- A **path** is a sequence of nodes and edges connecting a node with a descendant.
- An **ancestor** is a node reachable by repeated proceeding from child to parent. A **descendant** is a node reachable by repeated proceeding from parent to child. For example, if there's a path from node u to node v, then u is an ancestor of v and v is a descendant of u. If $u \neq v$, then u is a **proper ancestor** of v and v is a **proper descendant** of u.
- The **root** r of a tree is its top node known as the prime **ancestor**.
- The root of each subtree is said to be a **child** of the root r, and r is the **parent** of each subtree root.
- Two nodes u and v are **siblings** if they have the same parent.
- The **subtree** of a node, u, is the tree rooted at u and contains all of u's descendants.
- The **depth** of a node, u, is the length of the path from u to the root of the tree. This translates into the number of edges from node to the tree's root node. Thus, a root node will have a depth 0.
- The **height** of a node, u, is the length of the longest path from u to one of its descendants. It's defined as $h = max(height(u.left), height(u.right)) + 1$. Thus, the height of a tree is equal to the height of the root. Since leaves has no children, all leaves have height 0.
- A node, u, is a **leaf** if it has no children.

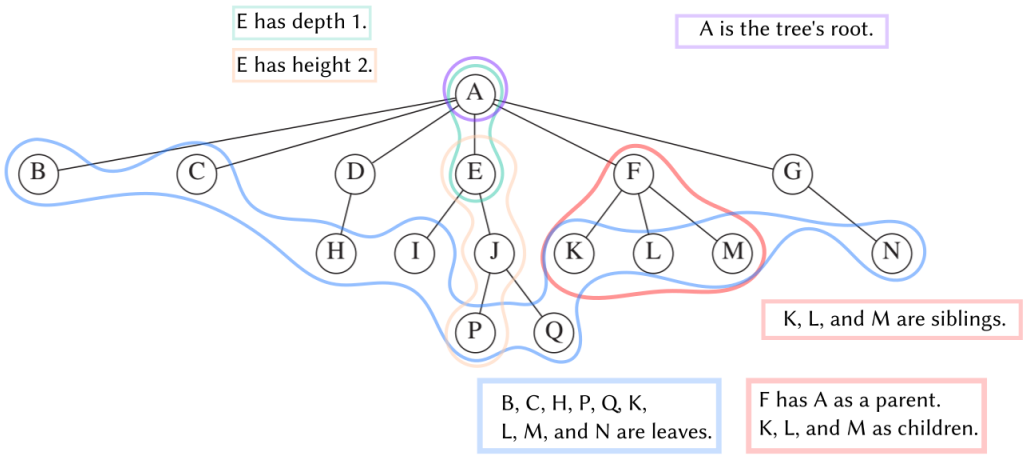


Figure 9: A tree

An m-ary tree is a rooted tree in which each node has no more than m children. When $m = 2$, the tree

is known as a **binary tree** (i.e., each node has at most two children, the left child and the right child). When $m = 3$, the tree is a **ternary tree**.

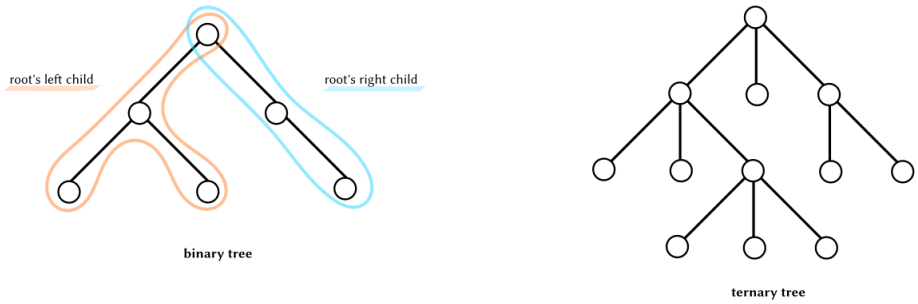


Figure 10: Binary and ternary tree

6.2 Implementation of trees

An obvious way to implement a tree would be to have in each node, besides its data, a link to each child of the node. However, the number of children per node can vary and is not known in advance which might translate into too much wasted space. The solution is simple: Keep the children of each in a linked list of nodes. In this way, each node can have as many children as the tree it composes allows.

```
template<typename Object>
struct TreeNode {
    Object element;
    TreeNode* firstChild;
    TreeNode* nextSibling;
}
```

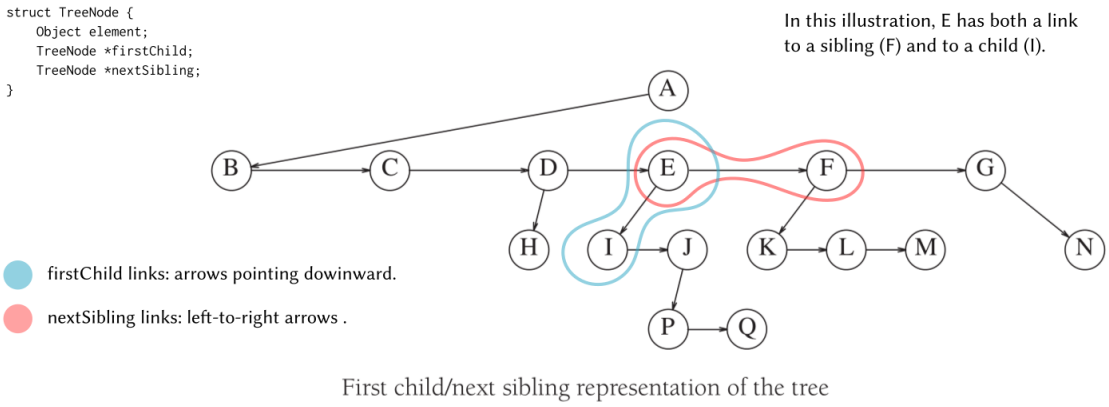


Figure 11: First child/next-sibling representation

6.3 Tree traversals

A **tree traversal** refers to the process of visiting each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

Trees can be traversed either:

- by deepening the search as much as possible on each child before going to the next sibling. This is known as **depth-first search** (DFS).

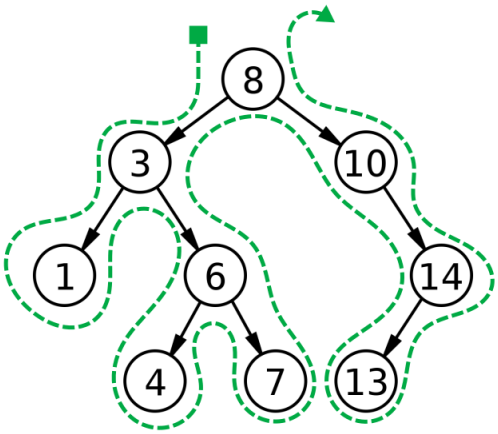


Figure 12: Depth-first search

- by visiting every node on a level before going to a lower level. This is known as **breadth-first**

search (BFS).

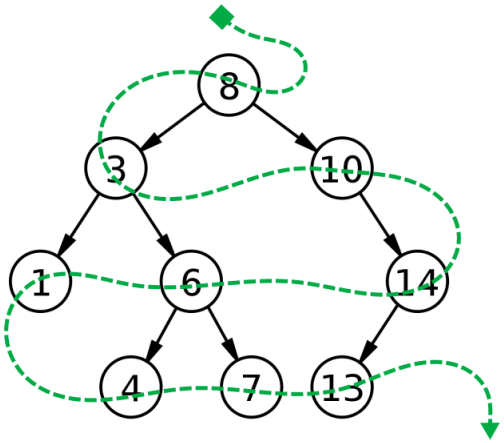


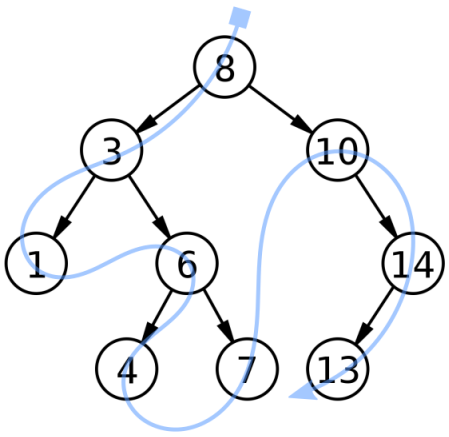
Figure 13: Breadth-first search

6.3.1 Depth-first search

Depending on when the work at a node is performed, a traversal of this type can be classified into:

- **pre-order traversal**. In a pre-order traversal, work at a node is performed before (*pre*) its children are processed.

> 1. Check if the current node is empty or null.
> 2. Display the data part of the root (or current node).
> 3. Traverse the left subtree by recursively calling the pre-order function.
> 4. Traverse the right subtree by recursively calling the pre-order function.



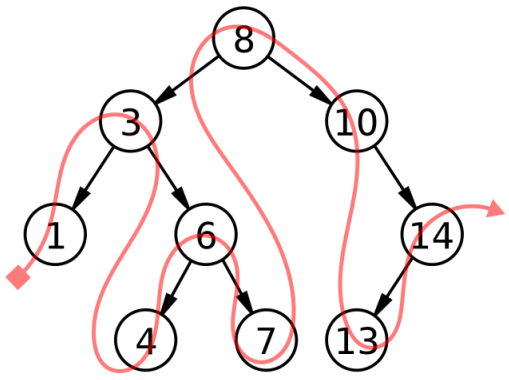

```
preorder(node)
    if (node == null)
        return
    visit(node)
    preorder(node.left)
    preorder(node.right)
```

8, 3, 1, 6, 4, 7, 10, 14, 13

Figure 14: Pre-order traversal

- **in-order traversal**. In an in-order traversal, work at a node is performed after its left child is visited, followed by the right child.

> 1. Check if the current node is empty or null.
> 2. Traverse the left subtree by recursively calling the in-order function.
> 3. Display the data part of the root (or current node).
> 4. Traverse the right subtree by recursively calling the in-order function.



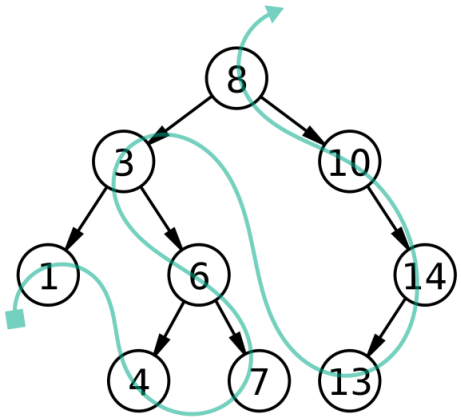

```
inorder(node)
    if (node == null)
        return
    inorder(node.left)
    visit(node)
    inorder(node.right)
```

1, 3, 4, 6, 7, 8, 10, 13, 14

Figure 15: In-order traversal

- **post-order traversal**. In a post-order traversal, work at a node is performed after (*post*) its children are evaluated.

1. Check if the current node is empty or null.
2. Traverse the left subtree by recursively calling the post-order function.
3. Traverse the right subtree by recursively calling the post-order function.
4. Display the data part of the root (or current node).




```
postorder(node)
    if (node == null)
        return
    postorder(node.left)
    postorder(node.right)
    visit(node)
```

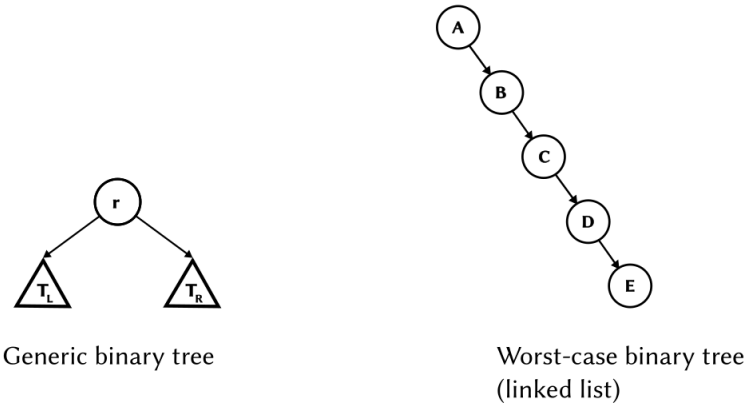
1, 4, 7, 6, 3, 13, 14, 10, 8

Figure 16: Post-order traversal

6.4 Binary trees

As stated earlier, a **binary tree** is a tree in which no node can have more than two children.

A property of a binary tree that is sometimes important is that the depth of an average binary tree is considerably smaller than N. An analysis shows that the average depth is $O(\sqrt{N})$, and that for a special type of binary tree, namely the **binary search tree**, the average value of the depth is $O(logN)$. Unfortunately, the depth can be as large as $N - 1$. In its worst case, we end up with a linked list (which is also a type of tree).



Generic binary tree

Worst-case binary tree
(linked list)

Figure 17: Generic binary tree and linked list

6.4.1 Implementation

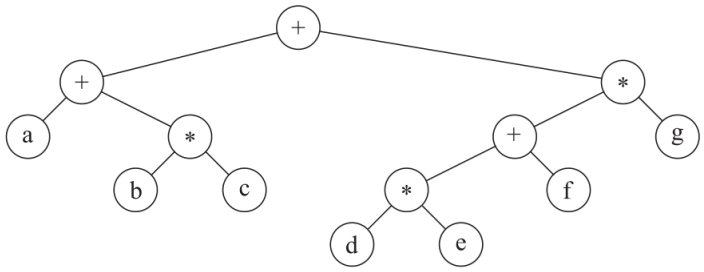
Given that a binary tree has at most two children per node, we can keep direct links to them. Thus, a binary node is just a structure consisting of the data plus two pointers to current node's left and right nodes. For example:

```
struct BinaryNode {
    Object element;    // data in the node
    BinaryNode* left;  // left child
    BinaryNode* right; // right children
}
```

Binary trees have many important uses not associated with searching. One of the principal uses of binary trees is in the area of compiler design (e.g., expression trees).

6.4.2 Expression trees

A **binary expression tree** is a specific kind of a binary tree used to represent expressions. The leaves of an expression are **operands**, such as constants or variable names, and the other nodes contain **operators**. We can evaluated an expression, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.



Expression tree for `(a + b * c) + ((d * e + f ) * g)`

Figure 18: Expression tree

- To produce an (overly parenthesized) *infix expression*, recursively produce parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This is an **in-order traversal**.
- To produce a *postfix expression*, recursively print out the left subtree, the right subtree, and then the operator. This is an **post-order traversal**.
- To produce a *prefix expression*, print out the operator first and then recursively print out the left and right subtrees. This is an **pre-order traversal**.

6.4.3 Constructing an expression tree

This algorithm describes the steps to convert a postfix expression into an expression tree:

```
ConvertPostfixToTree( EXPR ):
    stack ← []
    for EXPR → token:
        if token = operand:
            T ← Tree(root ← operand)
            stack.push(T)
        if token = operator:
            T1 ← stack.pop()
            T2 ← stack.pop()
            T ←  Tree(root ← operator, left ← T1, right ← T2)
            stack.push(T)
    return stack
```

Implementation: implementations/ExpressionTree

6.5 Binary search trees (BTSs)

A **binary search tree** is a special type of binary trees with the following property:

> For every node, X, in the tree, the values of all the items in its left subtree are smaller than the item in X, and the value of all the items in its right are larger than the item in X.

This implies that all the elements in the tree can be ordered in some consistent manner.

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that *each comparison allows the operations to skip about half of the tree*, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.

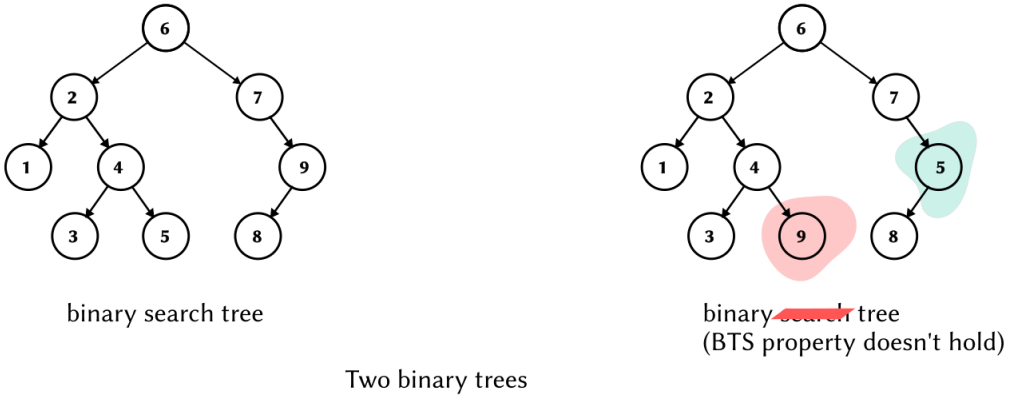


Figure 19: Binary search tree vs Non-binary search tree

Complete implementation: implementations/BinarySearchTree

6.5.1 Operations on a BTS

contains For this operation, we're required to return `true` if there is a node in the tree T that has item X. Otherwise, return `false`. If T is empty, then we can return `false` right away. Otherwise, if the item stored at T is X, we can return `true`. Otherwise, we can make a recursive call on a subtree of X, either left or right, depending on the relationship between X and the item stored in T. It's implied that the objects being compared have implemented the used relational operators appropriately.

```
Input: X is an item to search for and T is the node that roots
the subtree.
Output: Return true or false depending on whether the tree
contains the node storing the item.

contains( X, T ):
    if T = null:
        return false
    else if X < T.element():
        return contains X, T.left()
    else if X > T.element():
        return contains X, T.right()
    else:
        return true # Match
```

findMin and findMax These operations can either return the node (or its value) containing the smallest and largest element in the tree, respectively. To perform a `findMin`, start at the root and go left as long as there's a left child. The stopping point is the smallest element. The `findMax` routine is similar, except that branching is to the right child.

Recursive algorithm of `findMin`:

```
Input: the root of the tree.
Output: the smallest element in the tree.


findMin( T ):
    if T = null:
        return T
    if T.left() = null:
        return T;
    return findMin T.left()
```

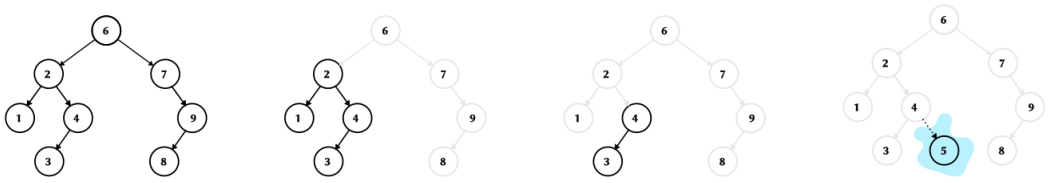
Iterative algorithm of `findMax`:

```
Input: the root of the tree.
Output: the largest element in the tree.

findMax( T ):
    if T ≠ null:
        until T.left() = null:
            T ← T.left()
    return T
```

insert To insert X into tree T, proceed down the tree as you would with the `contains` operation. If X is found, do nothing. Otherwise, insert X at the last spot on the path traversed.

Duplicates can be handled by keeping an extra field in the node object indicating the frequency of ocurrence for a particular element X. This adds some space extra space to the entire tree but is better than putting duplicates in the tree (which tends to make the tree very deep).

```
Input: X is an item to inserted in the tree T.
Output: void.

insert( X, T ):
    if T ← null:
        T ← BinaryNode(X, null, null)
    else if X < T.element()
        insert X, T.left()
    else if X > T.element()
        insert X, T.right()
    else:
        # duplicate; do nothing
```



Progression of inserting 5 into the tree

Figure 20: Progression of an insertion

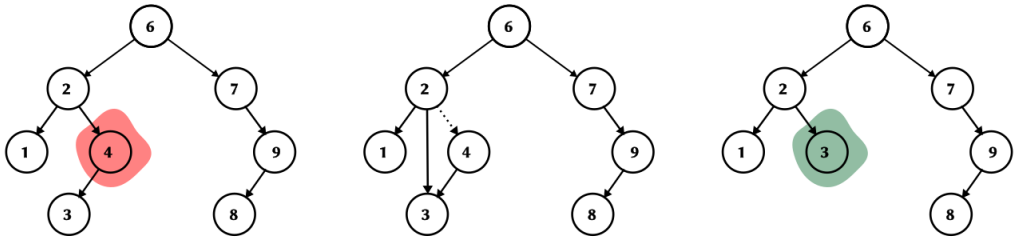
remove As is common with many data structures, the hardes operation is deletion. After finding the item to be deleted several possibilities must be considered:

- If the node is a leaf, it can be deleted immediately.

Removing a childless node (3) from the tree

Figure 21: Removing childless node from tree

- If the node has one child, the node can be deleted after ist parent adjusts a link to bypass the node.



Progression of removing node (4) with one child

Figure 22: Removing single child node from tree

- If the node has two children, replace the data of this node with the smallest data of the right subtree and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second `remove` is an easy one.

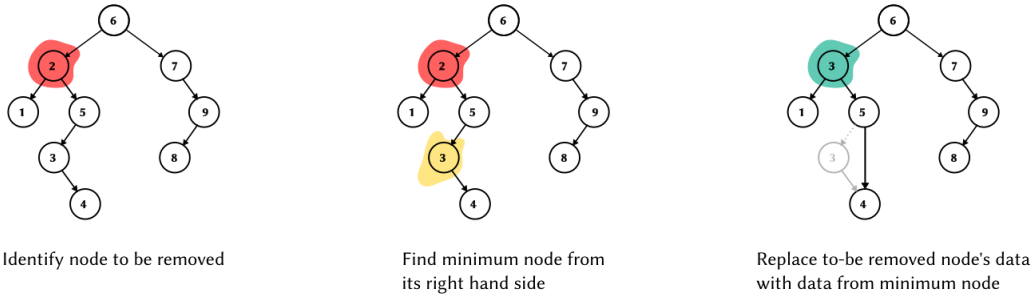


Figure 23: Removing node with two children from tree

```
Input: X is an item to be removed and T is the tree.
Output: void

remove( X, T ):
    if T = null:
        return; # Item not found; do nothing
    if X < T.element():
        remove X, T.left()
    else if X > T.element():
        remove X, T.right()
    else if T.left() ≠ null AND T.right() ≠ null:
        min ← findMin T.right()
        T.element ← min.element()
        remove T.element(), T.right()
    else:
        old-node ← T
        T ← T.left ≠ null ? T.left() : T.right()
```

The algorithm above is inefficient because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It's easy to remove this inefficiency by writing a special `removeMin` method.

If the number of deletions is expected to be small, then a popular strategy to use is **lazy evaluation**: When an element is to be deleted, it's left in the tree and merely *marked* as being deleted.

Pros of lazy evaluation (in this context):

- Easy to handle if a deleted item is reinserted, do not need to allocate a new node.
- Do not need to handle finding a replacement node.

Cons of lazy evaluation (in this context):

- The depth of the tree will increase. However this increase is usually a small amount relative to the size of the tree.

6.5.2 Average-case analysis

The running time of all the operations (except `makeEmpty` and copying) is $O(d)$, where d is the depth of the node containing the accessed item.

The sum of the depths of all nodes in a tree is known as the **internal path length**.

Let $D(N)$ be the internal path length for some tree T of N nodes. An N-node tree consists of an i-node left subtree and an $(N-i-1)$-node right subtree, plus a root at depth zero for $0 \leq i \leq N$. $D(i)$ is the internal path length of the left subtree with respect to its root. The same holds for the right subtree. We get the following recurrence $D(N) = D(i) + D(N-i-1) + N - 1$.

In a BTS all subtree sizes are equally likely, so

$$D(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} D(j)\right] + N - 1$$

By solving the above recurrence we show that $D(N) = O(NlogN)$. Thus the expected depth of any node is $O(logN)$.

Although it's tempting to say that this result implies the average running time of all the operatuons is $logN$, but this is not entirely true. The reason for this is that because of deletions, it's not clear that all BSTs are equally likely. In particular, the deletion algorithm favors making the left subtrees deeper than the right, because we're always replacing a deleted node with a node from the right subtree.

6.6 Balanced trees

If the input comes into a tree presorted, then a series of inserts will take *quadratic time* and give a very expensive implementation of a linked list, since the tree will consist only of nodes with no left children. One solution to the problem is to insist on an extra structural condition called **balance** where no node is allowed to get too deep.

There are quite a few general algorithms to implement **balanced trees**. Most are quite a bit more complicated than a standard binary search tree, and all take longer on average for updates. They do, however, provide protection against the embarrassingly simple cases. Below, we will sketch one of the oldest forms of balanced search trees, the AVL tree.

6.7 AVL trees

An AVL (Adelson-Velskii-Landis) tree is a a binary search tree that is *height-balanced*: At each node u, the height of the subtree rooted at `u.left` and the subtree rooted at `u.right` differ by at most 1. The balance condition must be easy to mantain, and it ensures that the depth of the tree $O(logN)$. The simplest idea is to require that the *left and right subtrees have the same height*.

It follows immediately that, if $F(h)$ is the minimum number of leaves in a tree of height h, then $F(h)$ obeys the Fibonacci recurrence $F(h) = F(h-1) + F(h-2)$ with base cases $F(0) = 1$ and $F(1) = 1$. This means $F(h)$ is approximately $\frac{\phi^h}{\sqrt{5}}$, where $\phi = (1+\sqrt{5})/2 \approx 1.61803399$ is the *golden ratio* (More precisely,$|\phi^h/\sqrt{5} - F(h)| \leq 1/2$.) This implies $h \leq log_\phi n \approx 1.44042008\log n$, although in practice is only slightly more than $\log N$.

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 at most 1. The height of an empty tree is defined to be -1.

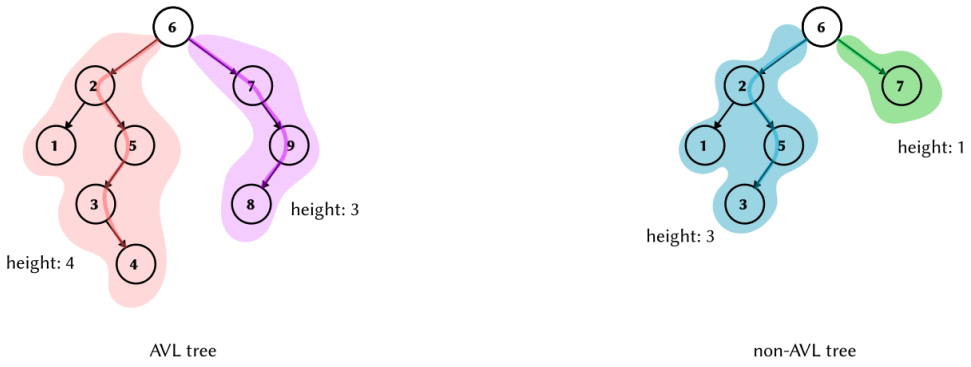


Figure 24: AVL tree and non-AVL tree

6.7.1 Insertion cases

After an insertion, only nodes that are on the path from the insertion point to the root might have their balance altered because only those nodes have their subtrees altered. As we follow the path up to the root and update the *balancing information*, we may find a fnode whose new balance violates the AVL condition.

Suppose that α is the first node on the path that needs to be rebalanced. Since any node has at most two children, and a height imbalance requires that the two subtrees's heights of α differ by two, it's easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

Cases 1 and 4 are mirror image symmetries with respect to α, as are cases 2 and 4. Thus, as a matter of theory, there are only two basic cases. Although programmatically, there are still four cases.

A **single rotation** of the tree fixes case 1 (and 4) in which the insertion occurs on the *outside* (i.e., left-left or right-right). A **double rotation** (which is slightly more complex) fixes case 2 (and 3) in which the insertion occurs on the *inside* (i.e., left-right or right-left). The implementation of a double rotation simply involves two calls to the routine implementing a single rotation, although conceptually it may be easier to consider them two separate and different operations.

Single rotation

NOTE: $k_1 < k_2$

Single rotation to fix case 1:

In the "Single rotation to fix case 1" figure, subtree X has grown to an extra level, causing it to be exactly two levels deeper than Z. The subtree Y cannot be at the same level as the X because then k_2 would have been out of balance *before* the insertion, and Y cannot be at the same level as Z because then k_1 would be the first node on the path toward the root that was in violation of the AVL balancing property.

To ideally rebalance the tree, we would like to move X up a level and Z down a level. By grabbing child node k_1 and letting the other nodes hang, the result is that k_1 will be the new root. The binary search tree property tells us that in the original tree $k_2 > k_1$, so k_2 becomes the right child of k_1 in the new tree. X and Z remain as the left child of k_1 and right child of k_2, respectively. Subtree Y, which holds items that are between k_1 and k_2 in the original tree, can be placed as k_2's left child in the new tree and satisfy all the ordering requirements.

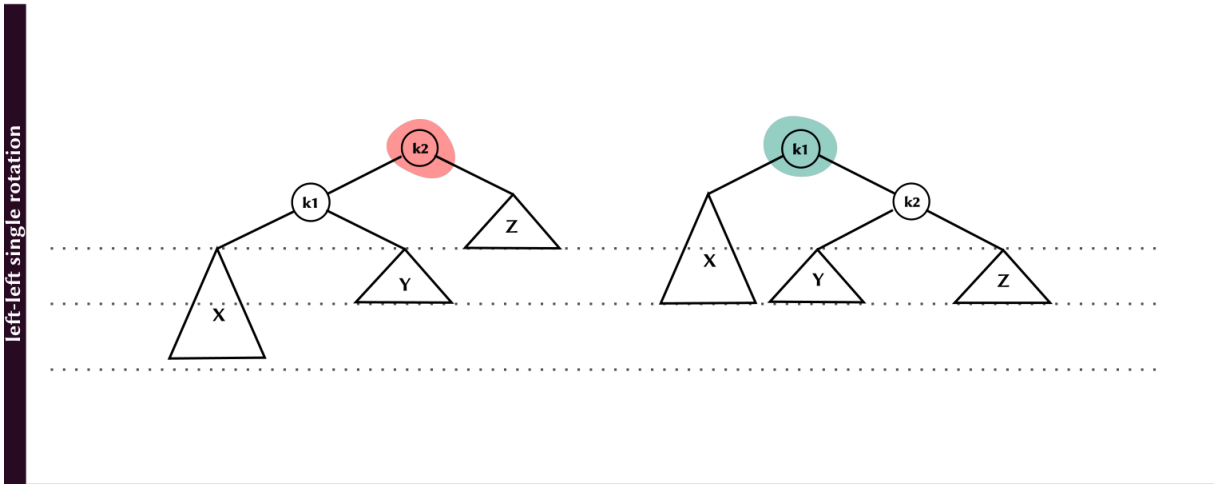


Figure 25: Single rotation to fix case 1

Simplified example Imagine we have this situation:

```
a
 \
  b
   \
    c
```

There's a height balance (i.e., the difference between his left and right subtree is greater than the balance factor, usually 1) in the node `a`. To fix this, we must perform a *left rotation* rooted at `a`. The steps are the following:

1. `b` becomes the new root.
2. `a` takes ownership of `b`'s left child as its right child. In this case, **`null`**.
3. `b` takes ownership of `a` as its left child.

The tree end up looking as follows:

```
   b
  / \
 a   c
```

Single rotation to fix case 4:

Case 4 represents a symmetric case and the "Single rotation to fix case 4" figure shows how a single rotation is applied.

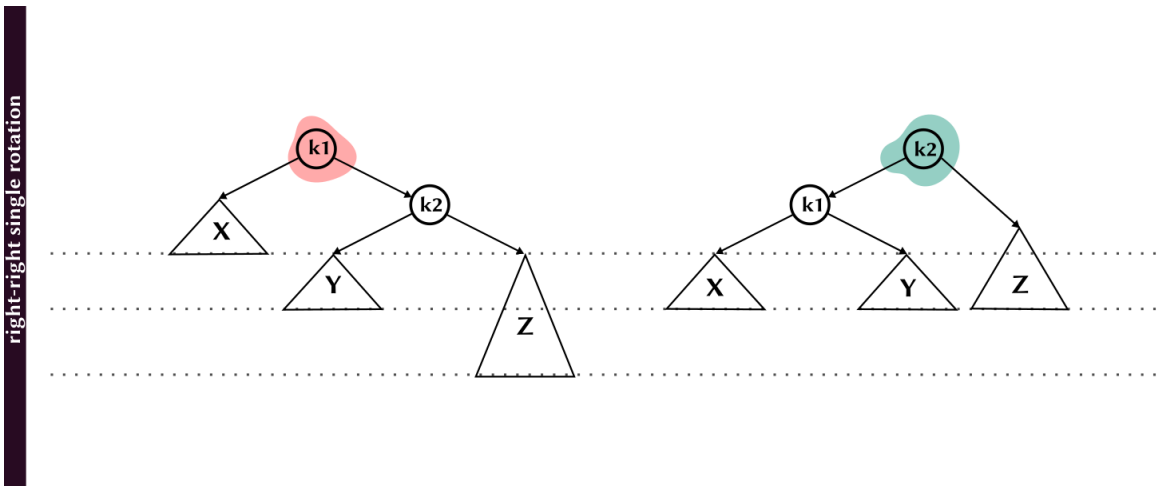


Figure 26: Single rotation to fix case 4

Simplified example Imagine we've this situation:

```
    c
   /
  b
 /
a
```

There's a height imbalance at node `c`, thus perform a *right rotation* rooted at `c`. The steps are the following:

1. `b` becomes the new root.
2. `c` takes ownership of `b`'s right child as its left child. In this case, **`null`**.
3. `b` takes ownership of `c` as its right child.

The tree ends up looking as follows:

```
  b
 / \
a   c
```

Example Let's suppose we start with an empty AVL tree and inserts the items $3, 2, 1$, and then 4 through 7 sequentially. As we insert certain items, we must do some rotations along the ways.

The first problem occurs when it's time to insert the item 1 because the AVL tree property is violated at the root. Thus, we perform a single rotation between the root and its left child to fix the problem. The "First rotation after violation of AVL property" figure shows the before and after the rotation.

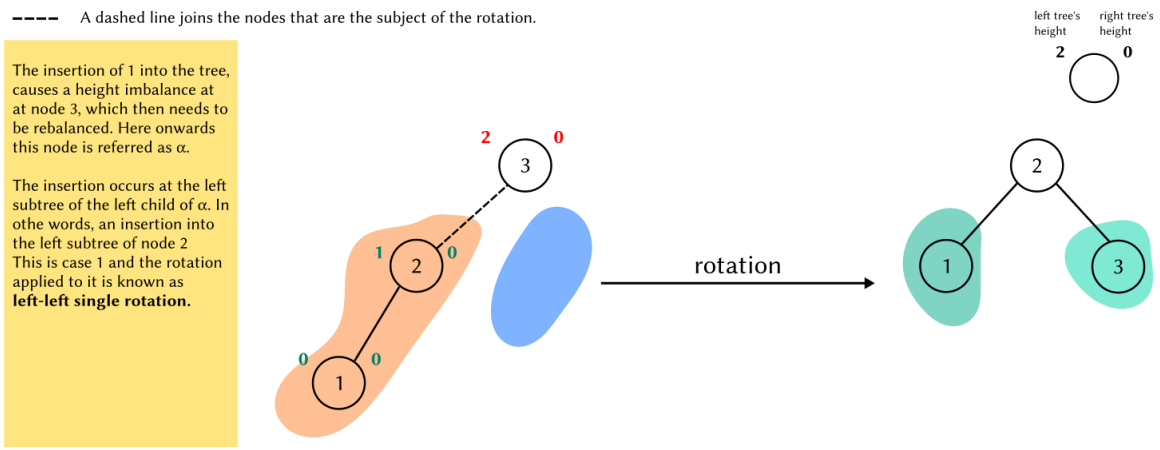


Figure 27: First rotation after violation of AVL property

Next we insert 4 but it causes no problems. However the insertion of 5 creates a violation at node 3 that is fixed by a single rotation. The rest of the tree has to be informed of this change so 2's right child must be reset to link to 4 instead of 3. The "Second rotation after violation of AVL property" figure shows the before and after the rotation.

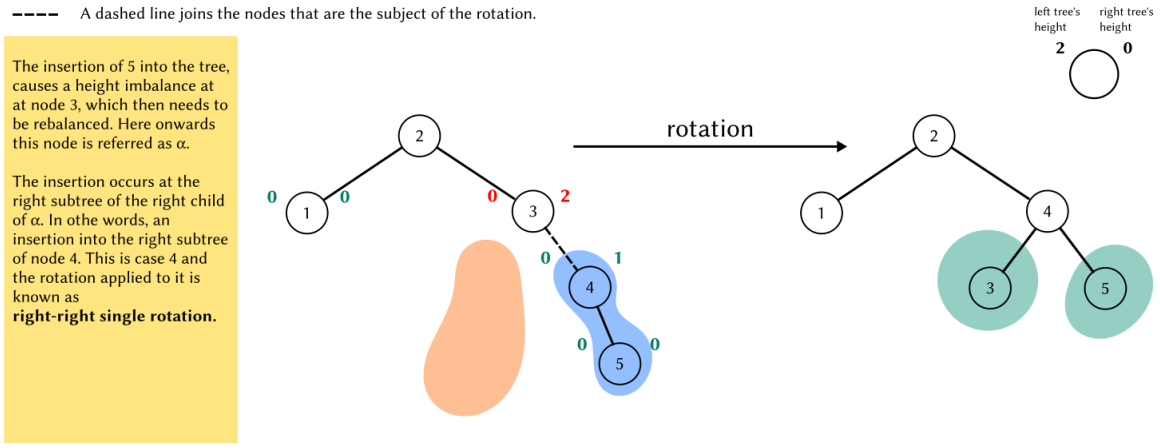


Figure 28: Second rotation after violation of AVL property

Next we insert 6 that causes a balance problem at the root, since its left subtree is of height 0 and its right subtree would be of height 2. Therefore, we perform a single rotation at the root between 2 and 4. The rotation is performed by making 2 a child of 4 and 4's original left subtree the new right subtree of 2. Every item in this subtree must lie between 2 and 4, so this transformation makes sense. The "Third rotation after violation of AVL property" figure shows the before and after the rotation.

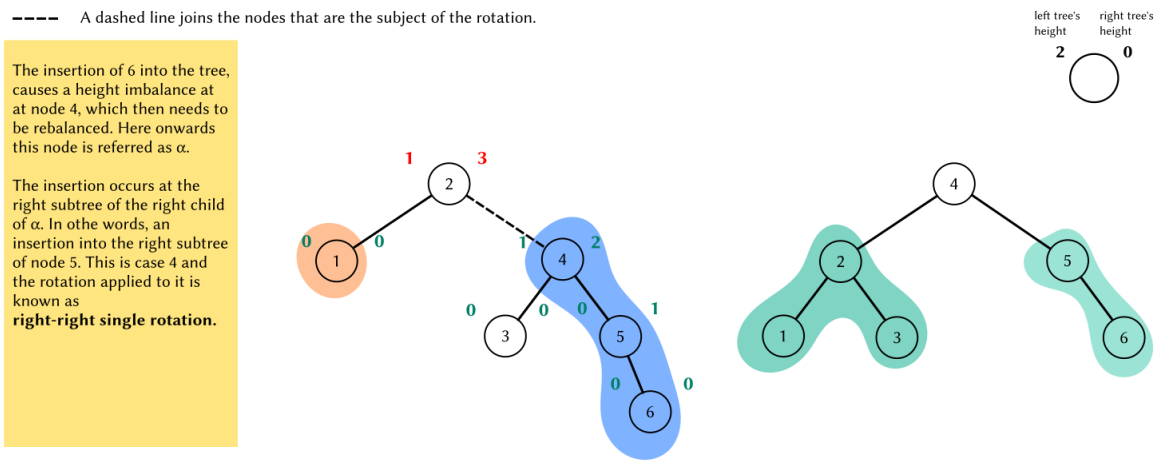


Figure 29: Third rotation after violation of AVL property

The next item we insert is 7, which causes another rotation. The "Fourth rotation after violation of AVL property" figure shows the before and after the rotation.

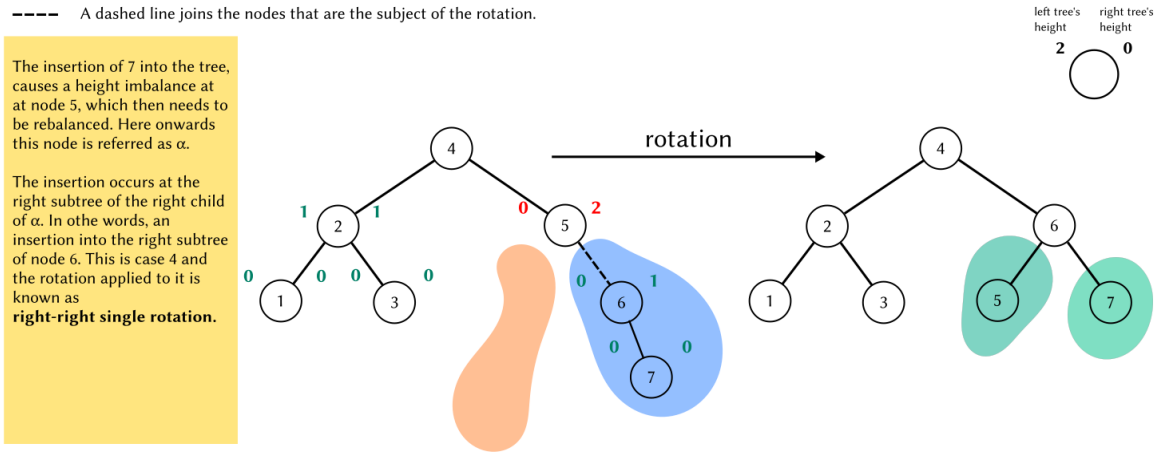


Figure 30: Fourth rotation after violation of AVL property

Double rotation The reason why single rotation doesn't work in cases 2 and 3 is because a subtree might be too deep, and a single rotation doesn't make it any less deep.

NOTE: $k_1 < k_2 < k_3$

Double rotation to fix case 2:

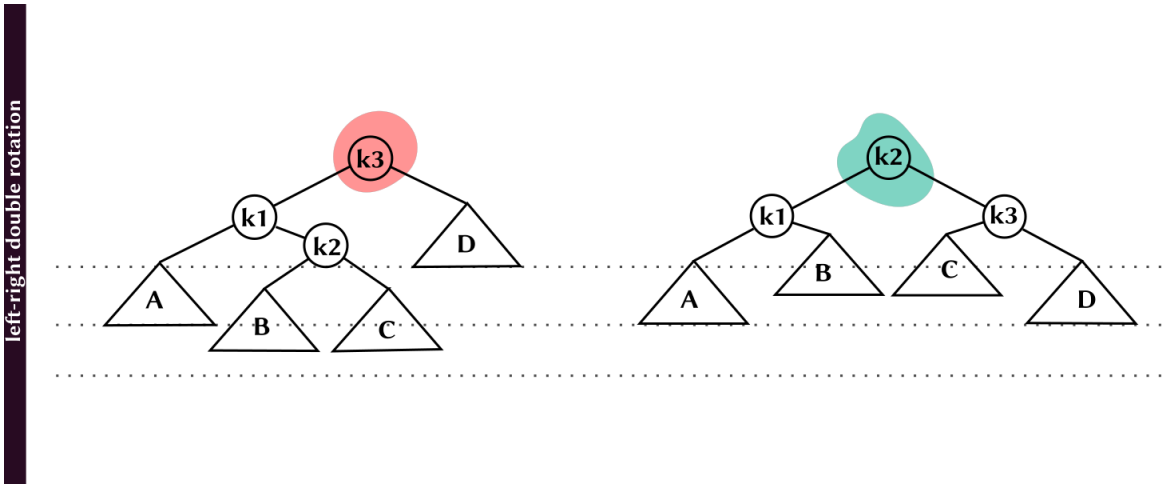


Figure 31: Left-right double rotation to fix case 2

Simplified example Imagine we've the following situation:

```
a
  \
   c
  /
b
```

There's a height balance at node a, however performing a single rotation won't achieve the desired result.

The steps are the following:

1. Perform a right rotation on the right subtree (we aren't rotating our current root).

```
   c
  /
b
```

becomes

```
b
  \
   c
```

2. Performing the rotation on the right subtree has prepared the root to be rotated left. The tree now is:

```
a
 \
  b
   \
    c
```

3. Rotate the tree left. The result is:

```
  b
 / \
a   c
```

Double rotation to fix case 3:

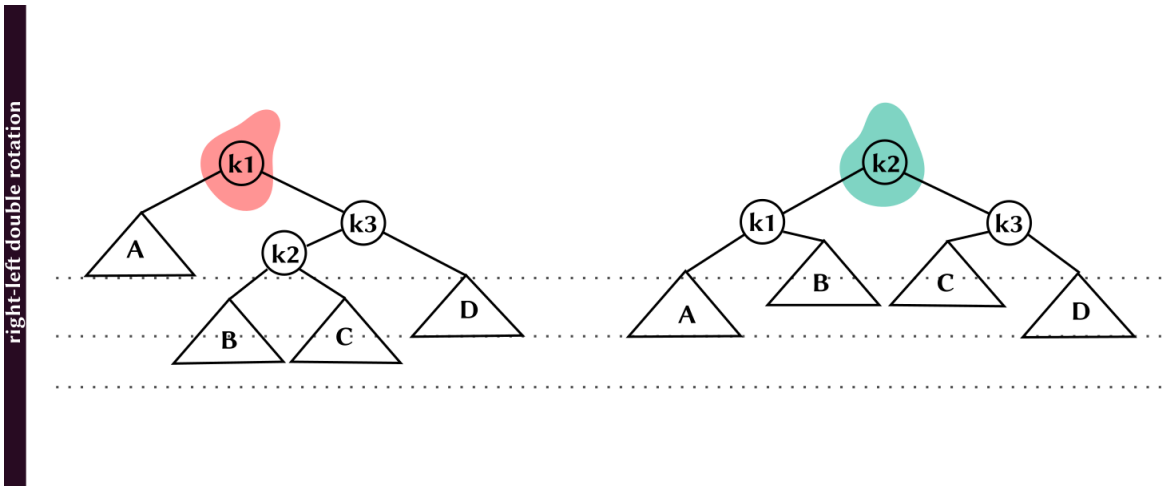


Figure 32: Right-left double rotation to fix case 3

Simplified example Take the following tree:

```
  c
 /
a
 \
  b
```

There's a height balance at node c.

The steps are the following:

1. Perform a left rotation on the left subtree. Doing so leaves us with this situation:

```
      c
     /
   b
  /
a
```

2. The tree can now be balanced using a single right rotation. Perform the right rotation at `c`. The result is:

```
   b
  / \
a     c
```

Example For this example, we'll continue with the AVL tree depicted in the "Fourth rotation after violation of AVL property" figure from the "Single rotation" section. We'll start by inserting 10 through 16 (in reverse order), followed by 8 and then 9.

Inserting 16 is easy since it doesn't violate the AVL balance property. However, inserting 15 causes a height imbalance at node 7, which is case 3 and solved by a right-left double rotation.

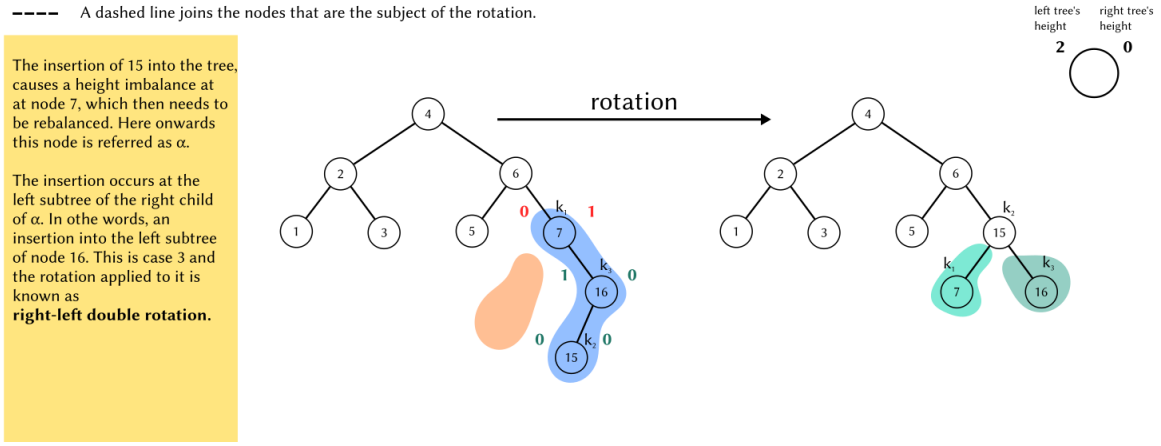


Figure 33: Fifth rotation after violation of AVL property

Next we insert 14, which also requires the same double rotation as before.

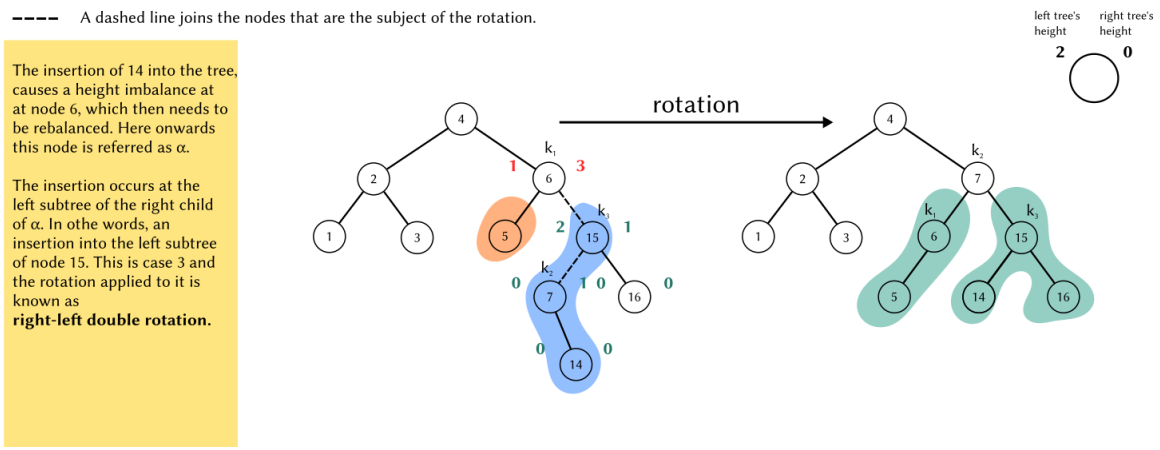


Figure 34: Sixth rotation after violation of AVL property

The insertions of 13, 12, 11, and 10 all cause height imbalance that are fixed using single rotations.

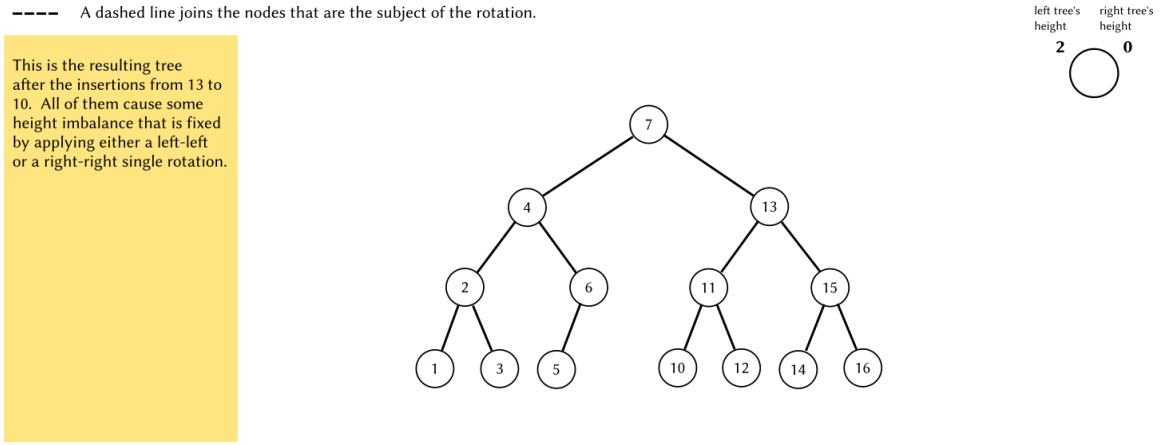


Figure 35: Tree after the seventh, eighth, ninth and tenth rotations after violation of AVL property

We can insert 8 without rotation, creating an almost perfectly balanced tree. The insertion of 9 causes a height imbalance at the node containing 10 which is fixed by a left-right double rotation.

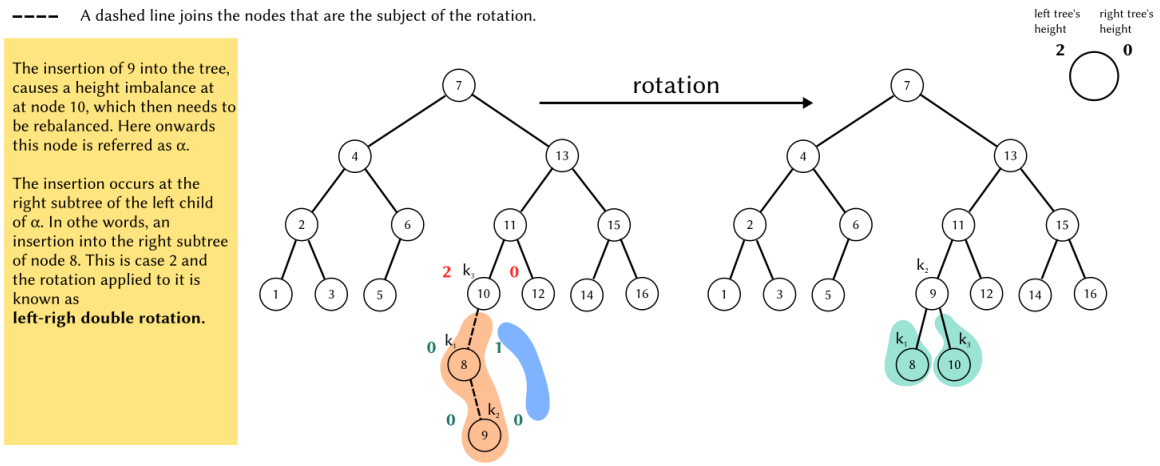


Figure 36: Tree after the twelfth rotation

6.7.2 When do you use rotations? Why?

Deciding when you need a tree rotation is usually easy, but determining which type of rotation you need requires a little thought.

A tree rotation is necessary when you have inserted or deleted a node which leaves the tree in an **unbalanced state**. An unbalanced state is defined as a state in which any subtree has a balance factor of greater than 1, or less than -1. That is, any tree with a difference between the heights of its two subtrees greater than 1, is considered unbalanced.

7 Lecture 7: AVL trees (cont.), splay trees, and B-trees

7.1 AVL tree implementation

implementation: implementations/AVLTree

7.1.1 Insertion

```
Input: X is the item to insert and T is the node that roots the subtree.

insert( X, T ):
    if T = null:
        T ← new AVLNode(X, null, null)
    else if X < T.item:
        insert(X, T.left)
    else if T.item < X:
        insert(X, T.right)

    balance(T)
```

7.1.2 Balance

```
Brief: Balance the subtree T by performing either a single or double
   rotation.
Input: T is the tree node to be rebalanced.

balance( T ):
    if T = null:
        return
    if T.left.height - T.right.height > 1:
        if T.left.left.height ≥ T.left.right.height:
            rotate-with-left-child(T)
        else:
            double-with-left-child(T)
    else if T.right.height - T.left.height > 1:
        if T.right.right.height ≥ T.right.left.height:
            rotate-with-right-child(T)
        else:
            double-with-right-child(T)

    T.height = max(T.left.height, T.right.height) + 1
```

7.1.3 Left-left single rotation (case 1)

```
Brief: Rotate binary tree with left child. This is a single rotation for
    case 1.
Input: K2 is the tree node that needs to be rebalanced.

rotate-with-left-child( K2 ):
    K1        = K2.left
    K2.left   = K1.right
    K1.right  = K2
    k2.height = max(K1.left.height, K2.right.height) + 1
    K1.height = max(K1.left.height, K2.height) + 1
    K2        = K1
```

7.1.4 Left-right double rotation (case 2)

```
Brief: Rotate binary tree with left child. This is a double rotation for
    case 2.
Input: K3 is the tree node that needs to be rebalanced.

double-with-left-child( K3 ):
    rotate-with-right-child(K3.left)
    rotate-with-left-child(K3)
```

7.1.5 Deletion

7.2 Amortized cost

Consider a sequence of M operations (insert/delete/find) and suppose the total cost is $O(M * f(N))$, irrespective of the actual sequence of operations; the average cost if $O(f(N))$ for each operation. This is called **amortized running time**. One caveat of this is that individual operations in the sequence can be expensive.

7.3 Splay trees

A **splay tree** is a tree with amortized time of $O(logN)$. It guarantees that any M consecutive tree operations starting from an empty tree take at most $O(MlogN)$ time (i.e., the tree has an $O(logN)$ amortized cost per operation).

The trick is to rebalance the tree after the `find()` operation. This is done by bringing the item return by `find()` to the tree's root while applying AVL rotations on the way to the root. The practical utility of

this method is that in many applications when a node is accessed, it is likely to be accessed again in the near future.

Furthermore, splay trees don't require the maintenance of height or balance information, thus saving space and simplifying the code to some extent.

7.3.1 A simple idea (that doesn't work)

One way to perform the restructing described above is by performing single rotations from the bottom up. This means that we rotate every node on the access path with its parent. For example, consider the following figures that show what happens after an access (a `find`) on k_1 in the a tree. The rotations have the effect of pushing k_1 all the way to the root making future accesses on k_1 easy but unfortunately, it pushed another node (k_3) almost as deep as k_1 used to be.

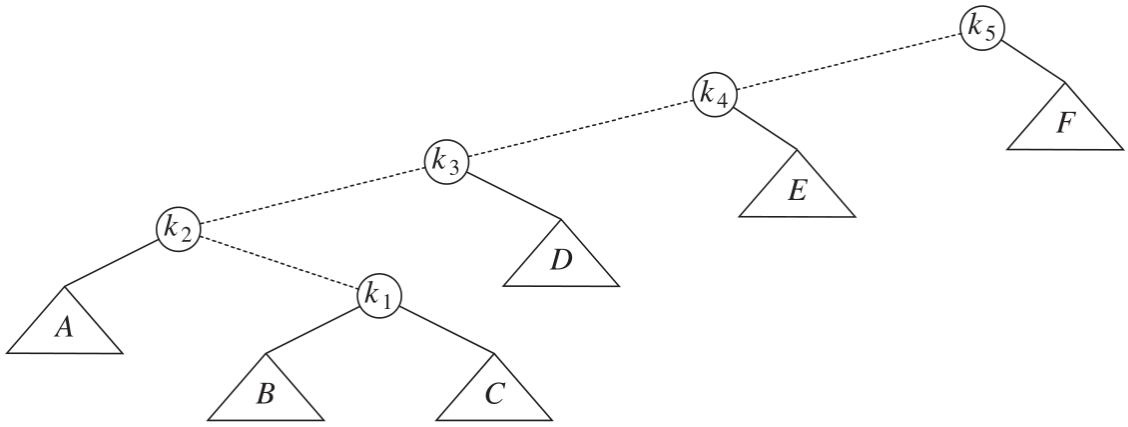


Figure 37: Before rotation: pushing k1 to the root

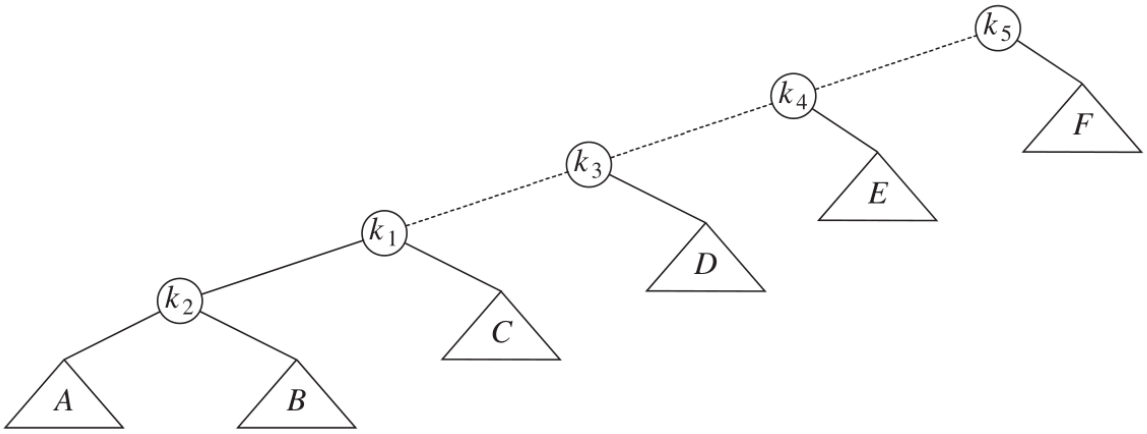


Figure 38: 1st rotation: pushing k1 to the root

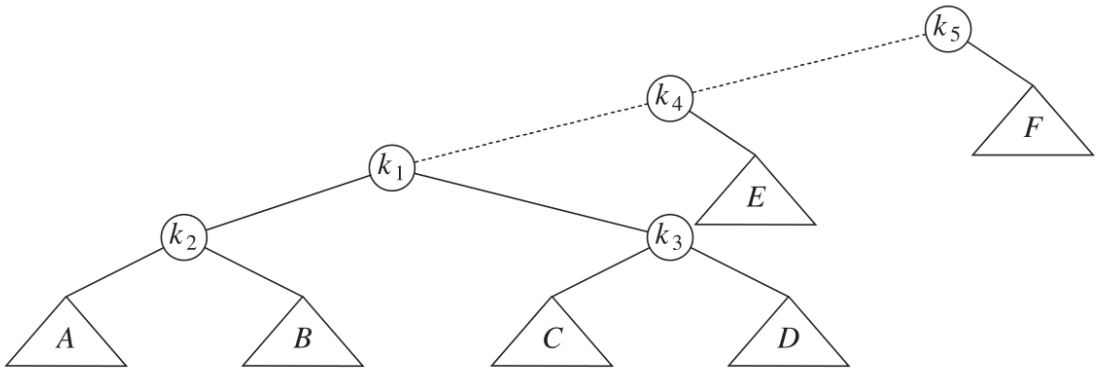


Figure 39: 2nd rotation: pushing k1 to the root

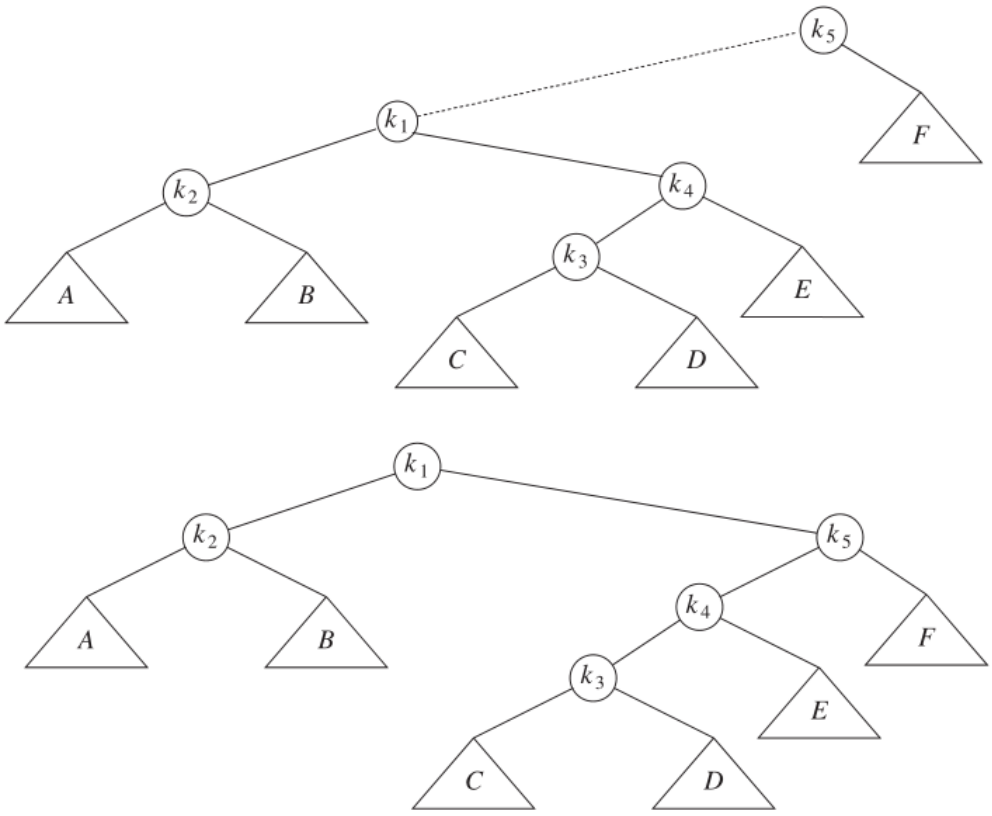


Figure 40: 3rd and 4th rotations: pushing k1 to the root

7.3.2 Splaying

The splaying strategy is similar to the rotation from the bottom up, except that we're a little more selective about how rotations are performed.

Let X be a (non-root) node on the access path at which we are rotating. If the parent of X is the root of the tree, we merely rotate X and the root. This is the last rotation along the access path. Otherwise, X has both a parent (P) and a grandparent (G), and there are two cases, plus symmetries, to consider:

- The first case is the **zig-zag** case. Here X is a right child and P is a left child (or vice versa). If this is the case, we perform a double rotation, exactly like an AVL double rotation.

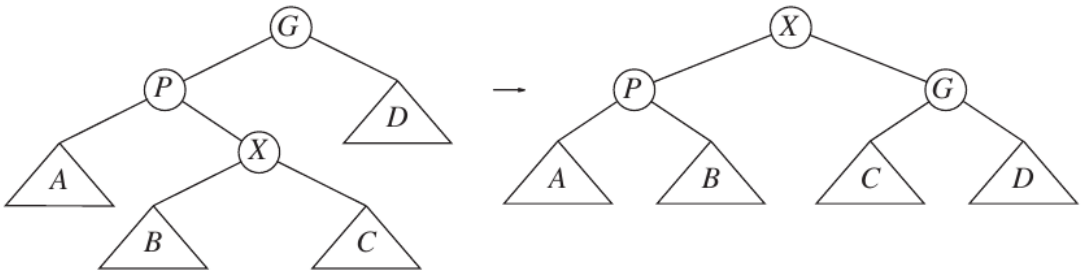


Figure 41: Zig-Zag

- The second case is a **zig-zig** case: X and P are both left children (or, in the symmetric case, both right children).

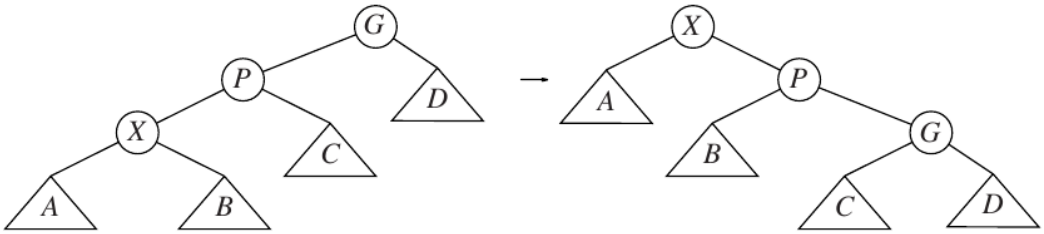


Figure 42: Zig-Zig

7.3.3 Fundamental property of splay trees

- When access paths are long, they lead to longer-than normal search time.
- When accesses are cheap, the rotations are not as good and can be bad.
- The extreme case is the initial tree formed by the insertions.

7.3.4 Summary

8 The analysis of splay trees is difficult, because it must take into account the ever-changing structure of the tree.

- Splay trees are much simpler to program than most balanced search trees, since there are fewer cases to consider and no balance information to maintain.
- Some empirical evidence suggests that this translates into faster code in practice, although the case for this is far from complete.

- There are several variations of splay trees that can perform even better in practice.

7.4 B-trees

A **B-tree** is a generalization of a binary tree, which is efficient in the *external memory model*.

So far, we have assumed that we can store an entire data structure in the main memory of a computer. Now imagine the scenario of a tree that is immensely huge and can't fit into memory. Now the data structure must reside on the hard, which changes the rules of the game, because the Big-Oh model is no longer meaningful. The problem is that a Big-Oh analysis assumes that all operations are equal. However, this is not true, especially when disk I/O is involved.

Here is how the typical search tree performs on disk: Suppose we want to access the driving records for citizens in the state of Florida. We assume that we have 10,000,000 items, that each key is 32 bytes (representing a name), and that a record is 256 bytes. We assume this does not fit in main memory and that we are 1 of 20 users on a system (so we have 1/20 of the resources). Thus, in 1 sec we can execute many millions of instructions or perform six disk accesses.

The first idea might be to use a unbalanced binary search tree but it turns into a disaster. In the worst case, it has linear depth and thus could require 10,000,000 disk accesses. On average, a successful search would require $1.38logN$ disk accesses, and since $log10000000 \approx 24$, an average search would require 32 disk accesses, or 5 sec.

In a typical randomly constructed tree, we would expect that a few nodes are three times deeper; these would require about 100 disk accesses, or 16 sec.

An AVL tree is somewhat better. The worst case of $1.44logN$ is unlikely to occur, and the typical case is very close to log N. Thus an AVL tree would use about 25 disk accesses on average, requiring 4 sec.

We want to reduce the number of disk accesses to a very small constant, such as three or four. Basically, we need smaller trees (i.e., trees with more branching and thus less height). An M**-ary search tree** allows M-way branching. *As branching increases, the depth decreases*. Whereas a complete binary tree has height that is roughly $log_{2}N, a complete M-ary tree has height that is roughly $log_{M}N$.

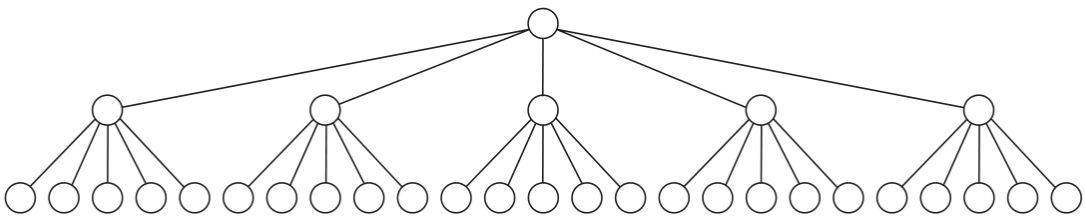


Figure 43: 5-ary tree with 31 nodes

A **B-tree** of order M is an M-ary tree with the following properties:

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M-1$ keys to guide the searching; key i represents the smallest key in subtree $i+1$.
3. The root is either a leaf or has between two and M children.
4. All nonleaf nodes (except the root) have between $M/2$ and M children.
5. All leaves are at the same depth and have between $L/2$ and L data items, for some L.

M and L are determined based on disk block (one access should load a whole node).

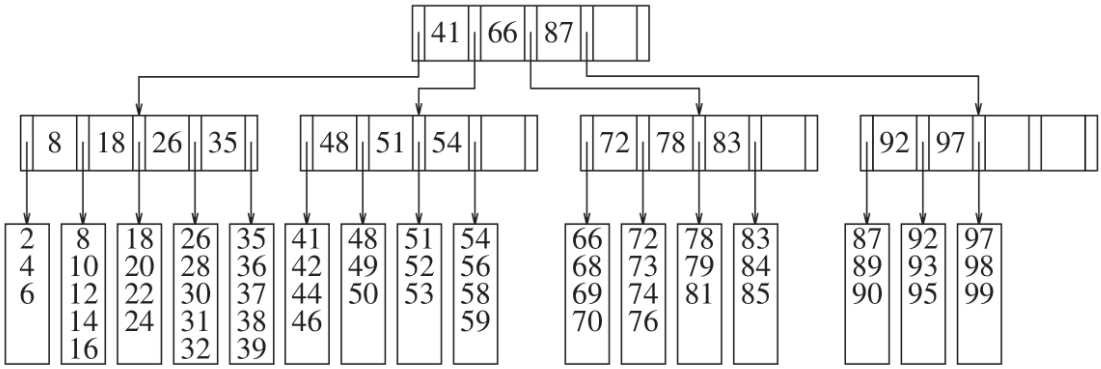


Figure 44: B-tree of order 5

The "B-tree of order 5" figure depicts a B-tree of order 5:

- Each represents a disk block.
- All nonleaf nodes have between 3 and 5 children.
- $L = M = 5$, however this not need to be the case.
- Since $L = 5$, each leaf has between 3 and 5 data items.

In the Florida driving records example:

- A single block is supposed to hold $8,192$ bytes.
- Each key uses 32 bytes. In a B-tree of order M, we would've $M-1$ keys, for a total of $32\times(M-1) = 32M-32$ bytes, plus M branches.
- Each branch is essentially a number of another disk block, so we can assume a branch is 4 bytes. Thus the branches use $4M$ bytes.
- The total memory requirement for a nonleaf node is thus $32M-32+4M = 36M-32$.
- Since $8192 \leq 36M-32$, the largest value of M for which this is no more thant 8192 is 228. Thus we choose $M = 228$.

- Since each data record is 256 bytes, we'd be able to fit 32 records in a block. Thus we'd choose $L = 32$.
- Each leaf has between 16 and 32 data records and each internal node (except the root) branches in at least 114 ways.
- Since there are $10,000,000$ records, there are, at most, $625,000$ leaves. Consequently, in the worst case, leaves would be on level 4 of the tree.
  - The worst-case number of accesses is given by approximately $log_{M/2}N$.
  - The root and the next level could be *cached* in main memory, so that over the long run, disk accesses would be needed only for level tree and deeper.

7.4.1 Example

7.4.2 Insertion

- Put it in the appropriate leaf.
- If the leaf is full, break it in two, adding a child to the parent.
- If this puts the parent over the limit, split upwards recursively.
- If you need to split the root, add a new one with two children. This is the only way you add depth.

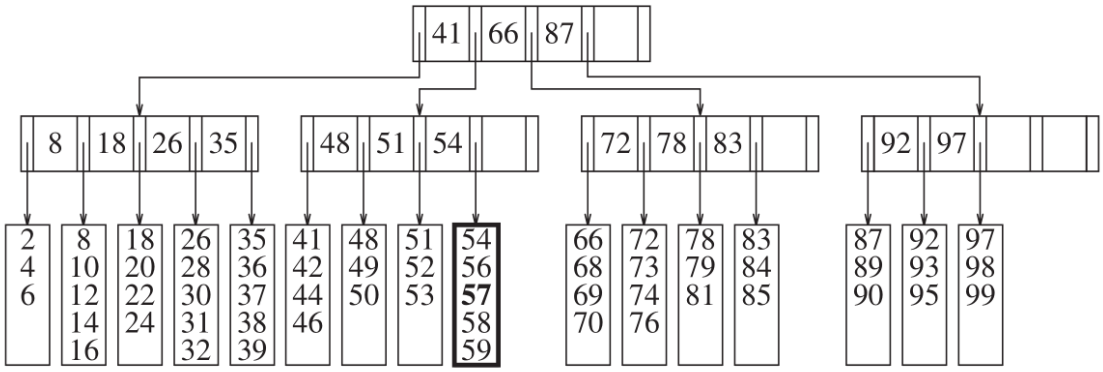


Figure 45: B-tree of order 5: Inserting 57

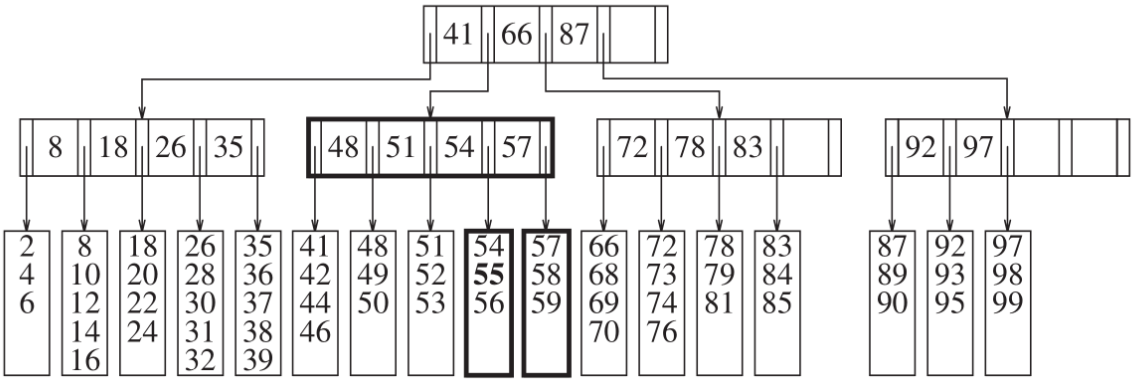


Figure 46: B-tree of order 5: Inserting 55

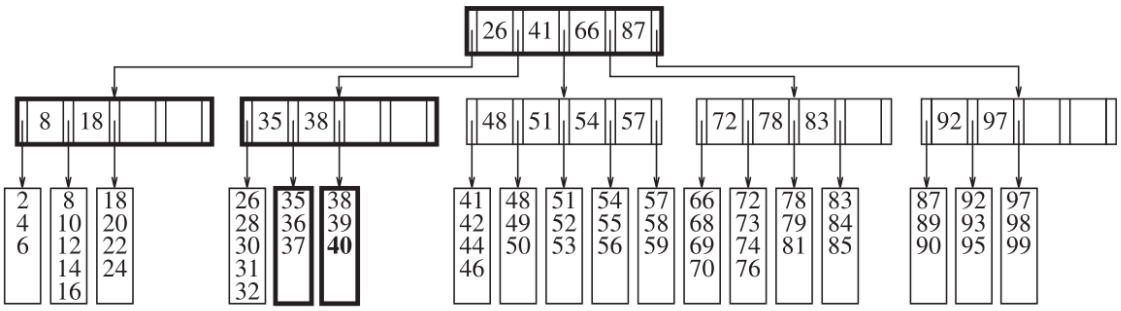


Figure 47: B-tree of order 5: Inserting 40

7.4.3 Deletion

- Delete from appropriate leaf.
- If the leaf is below its minimum, adopt from a neighbor if possible.
- If that's not possible, you can merge with the neighbor. This causes the parent to lose a branch and you continue upward recursively.

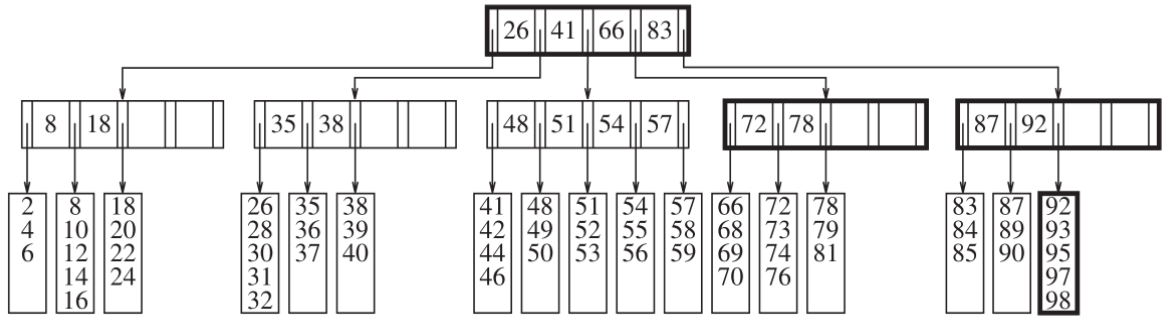


Figure 48: B-tree of order 5: Inserting 40

8 Lecture 08: Sets and maps

8.1 STL containers

The *vector* and *list* containers are inefficient for search and insert. Thus, the STL provides the *set* and *map* containers which guarantee logarithmic time for insertion, deletion and searching.

8.2 The set container

A *set* is an ordered container that doesn't allow duplicates:

- It stores objects of type `Key` in sorted order.
- The `Value` is the `Key` itself, without any additional data.

8.2.1 Insertion

The `insert` method returns an `iterator` which represents the position of the newly inserted item or the existing item that caused the insertion to fail (i.e., no duplicates).

The STL defines a class template called `pair` that is little more than a struct with members first and second to access the two items in the pair. There are two different `insert` routines:

- `pair<iterator,bool> insert( const Object & x );`
- `pair<iterator,bool> insert( iterator hint, const Object & x );`

`pair<T1, T2` is a heterogenous pair; it holds one object of type `T1` and another of type `T2`. For example:

```
pair<bool, double> result; // a pair with key as boolean and value as
    double
result.first = true; // the method first access the pair's key
result.second = 2.5; // the method second accesses the pair's value
```

8.2.2 Insertion with hint

The two-parameter `insert` allows the specification of a hint, which represents the position where `x` should go. If the hint is accurate, the insertion is fast, often $O(1)$. If not, the insertion is done using the normal insertion algorithm and performs comparably with the one-parameter insert. For instance, the following code might be faster using the two-parameter `insert` rather than the one-parameter `insert`:

```
set<int> s;
for (int i = 0; i < 10; i++) {
    const auto result = s.insert(s.end(), i);
}
```

8.2.3 erase

There are several versions of `erase`:

- `int erase( const Object & x );`: This version removes `x` (if found) and returns the number of items actually removed, which is obviously either 0 or 1.
- `iterator erase( iterator itr );`: This one behaves the same as in `vector` and `list`, and it removes the object at the position given by the iterator, returns an iterator representing the element that followed `itr` immediately prior to the call to erase, and invalidates `itr`, which becomes stale.
- `iterator erase( iterator start, iterator end );`: This one behaves the same as in a `vector` or `list`, removing all the items starting at `start`, up to but not including the item at `end`.

8.2.4 find

The set provides a `find` routine that returns an iterator representing the location of the item (or the endmarker if the search fails). This provides considerably more information, at no cost in running time. The signature of `find` is iterator `find( const Object & x )const;`.

By default, ordering uses the `less<Object>` function object, which itself is implemented by invoking `operator<` for the `Object`. An alternative ordering can be specified by instantiating the `set` template

with a function object type. For instance, we can create a `set` that stores string objects, ignoring case distinctions by using the `CaseInsensitiveCompare` function object:

```
class CaseInsensitiveCompare {
    public:
        bool operator( )( const string & lhs, const string & rhs ) const {
            return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0;
        }
};

set<string,CaseInsensitiveCompare> s;
s.insert("Hello");
s.insert("HeLLo");
std::cout << "The size is: " << s.size( ) << std::endl; // size is 1
```

8.3 The map container

A `map` is used to store a collection of ordered entries that consists of keys and their values:

- Keys must be unique, but several keys can map to the same values. Thus values need not be unique.
- The keys in the `map` are maintained in logically sorted order.

The `map` also supports `insert`, `find`, `erase`, `begin`, `end`, `size`, and `empty`. For `insert`, one must provide a `pair<KeyType,ValueType>` object. Although `find` requires only a key, the `iterator` it returns references a `pair`.

There's another operation that yields simple syntax. The array-indexing operator is overloaded for maps as follows:

```
ValueType& operator[] ( const KeyType & key );
```

The semantics of `operator[]` are as follows:

- If `key` is present in the `map`, a reference to the corresponding value is returned.
- If `key` is not present in the map, it is inserted with a default value into the map and then a reference to the inserted default value is returned. The default value is obtained by applying a zero-parameter constructor or is zero for the primitive types.

These semantics do not allow an accessor version of `operator[]`, so `operator[]` cannot be used on a `map` that is constant. For instance, if a map is passed by constant reference, inside the routine, `operator[]` is unusable.

8.3.1 Example

```
#include <iostream>
#include <string>
#include <map>

int main() {
  std::map<std::string, double> salaries;

  salaries["Pat"] = 78000.0;

  std::cout << salaries["Pat"] << std::endl;
  std::cout << salaries["Per"] << std::endl;

  std::map<std::string, double>::const_iterator itr;
  itr = salaries.find("Paul");

  if (itr == salaries.end()) {
    std::cout << "Not an employee" << std::endl;
  }
  else {
    std::cout << itr->second << std::endl;
  }

  return 0;
}
```

8.4 Implementation of set/map in C++

C++ requires that set and map support the basic `insert`, `erase`, and `find` operations in logarithmic worst-case time. The underlying implementation is a **balanced binary search tree**.

An important issue in implementing `set` and `map` is providing support for the `iterator` classes. Of course, internally, the `iterator` maintains a pointer to the "current" node in the iteration. The hard part is efficiently advancing to the next node. There are several possible solutions, but the smart solution is to use **threaded** trees which are used throughout different implementations in the STL.

This solution involves mantaining the extra links only for nodes that have `nullptr` left or right links by using extra `Boolean` variables to allow the routines to tell if a left link is being used as a standard binary search tree left link or a link to the next smaller node, and similarly for the right link.

8.4.1 Threaded binary trees

A binary tree is threaded by

- making all right child pointers (that would normally be `nullptr`) point to the in-order successor of the node (if it exists), and
- making all left child pointers (that would normally be `nullptr`) point to the in-order predecessor of the node.

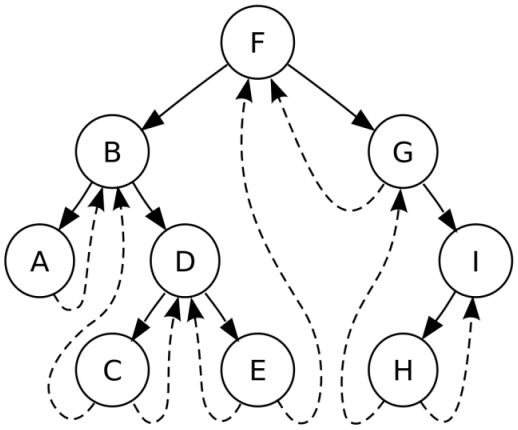


Figure 49: Threaded tree, special threaded links are dashed arrows

In this type of tree, we have the pointers reference the next node in a in-order traversal. These are known as threads.

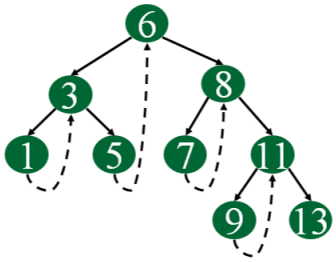
We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer.

Why do we need threaded binary trees?

- Binary trees have a lot of wasted space.
- Threaded binary trees make the tree traversal faster since we don't need a stack or use recursion for traversal.

8.4.2 Types of threaded binary trees

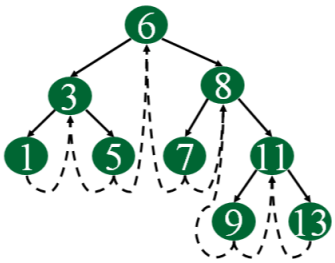
- **Single threaded:** Each node is threaded towards either the in-order predecessor o successor (left or right). This means all right null pointers will point to the in-order successor **or** all left null pointers will point to the in-order predecessor.



Single Threaded Binary Tree

Figure 50: Single threaded binary tree

- **Double threaded:** Each node is threaded towards both the in-order predecessor and successor (left and right). This means that all right null pointers will point to the in-order successor **and** all left null pointers will point to the in-order predecessor.



Double Threaded Binary Tree

Figure 51: Double threaded binary tree

8.4.3 An example

We would like to write a program to find all words that can be changed into at least 15 other words by a single one-character substitution. We assume that we have a dictionary consisting of approximately 89,000 different words of varying lengths. Most words are between 6 and 11 characters.

First approach: The most straightforward strategy is to use a `map` in which the keys are words and

the values are vectors containing the words that can be changed from the key with a one-character substitution.

```
/*
@brief The function prints the contents of the map only for the elements
for which the vector of strings has size greater than or equal to
   min_words.
@param adjacentWords input map from string to vector of strings.
@param min_words minimu number of words to consider printing.
*/
void print_high_changeables(
    const std::mapt<std::string>, std::vector<std::vector>> adjacent_words
       ,
    int min_words = 15
) {
    for (auto& entry : adjacent_words) {
        const std::vector<std::string>& words = entry.second;
        if (words.size() >= min_ords) {
            std::cout << entry.first << "(" << words.size() << "): ";
            for (auto& str : words) {
                std::cout << " " << str << std::endl;
            }
        }
    }
}
```

The function above only shows how to print the the `map` that is eventually produced. How do we construct the `map` though?

One way is to test if two words are identicall except for a one-character substitution. Then, we can brute-force our way testing all pairs of words and constructing the `map`. If we find a pair of words that differ in only one character, we can update the `map` by adding this new word (that differs by a single character) to the `map`.

```
/*
@brief Check if two words differ by a single character.
@return bool
*/
bool nearly_equal( const std::string& word1, const std::string& word2 ) {
    if (word1.length() != word2.length()) return false;
    int diffs = 0;
    for (int i = 0; i < word1.length(); i++) {
        if (word1[i] != word2[i]) {
            if (++diffs > 1) return false;
        }
    }
    return diffs == 1;
}

std::map<std::string, std::vector<std::string>> compute_adjacent_words(
    const std::vector<std::string>& words
) {
    std::map<std::string, std::vector<std::string>> adj_words;
    for (int i = 0; i < words.size(); i++) {
        for (int j = i; j < words.size(); j++) {
            adj_words[ words[i] ].push_back(words[j]);
            adj_words[ words[j] ].push_back(words[i]);
        }
    }
    return adj_words;
}
```

Second approach:

The problem with the algorithm above is that it's slow. An obvious improvement is to avoid comparing words of different lenghts. We can do this by grouping words by their length, and then running the previous algorithm on each of the separate groups. To do this, we can use a second `map` where the key is an integer representing a word length, and the value is a collection of all words of that length.

```
/*
@brief Compute a map in which the keys are words and values are
vectors of words that differ in only one character from the
corresponding key. Uses a quadratic algorithm, but speeds things
up a little by maintaining an additional map that groups words
by their length.
*/
std::map<std::string, std::vector<std::string>> compute_adjacent_words(
    const std::vector<std::string>& words
) {
    std::map<std::string, std::vector<std::string>> adj_words;
    std::map<int, std::vector<std::string>> words_by_length;

    // group the words by their length
    for (auto& word : words) {
        words_by_length[ word.length() ].push_back(word);
    }

    // work on each group separately
    for (auto& entry : words_by_length) {
        const std::vector<std::string>& groups_words = entry.second;

        for (int i = 0; i < groups_words.size(); i++) {
            for (int j = i + 1; j < groups_words.size(); j++) {
                if (nearly_equal(groups_words[i], groups_words[j])) {
                    adj_words[ groups_words[i] ].push_back( groups_words[j
                        ] );
                    adj_words[ groups_words[j] ].push_back( groups_words[i
                        ] );
                }
            }
        }
    }

    return adj_words;
}
```

Third approach:

This third algorithm is more complex and uses additional `map`s. As before, words are grouped by lengths, and then each group is worked on separately.

```
std::map<std::string, std::vector<std::string>> compute_adjacent_words(
    const std::vector<std::string>& words
) {

    std::map<std::string, std::vector<std::string>> adj_words;
    std::map<int, std::vector<std::string>> words_by_length;

    // group words by their length
    for (auto& word : words) {
        word_by_length[ word.length() ].push_bacj(word);
    }

    // work on each group separately
    for (auto& entry : words_by_length) {
        const std::vector<std::string>& groups_words = entry.second;
        int group_num = entry.first;

        // work on each position in each group
        for (int i = 0; i < group_num; i++) {
            std::map<std::string, std::vector<std::string>> rep_to_word;

            // Remove one character in specified position, computing
               representative.
            // Words with same representatives are adjacent; so populate a
                map ...
            for (auto& word : groups_words) {
                std::string rep = word;
                rep.erase(i, 1);
                rep_to_word[ rep ].push_back(word);
            }

            // ... and then look for map values with more than one string
            for (auto& entry : rep_to_word) {
                const std::vector<std::string>& clique = entry.second;
                if (clique.size() >= 2) {
                    for (int p = 0; p < clique.size(); p++) {
                        for (int q = p + 1; q < clique.size(); q++) {
                            adj_words[ clique[p] ].push_back(clique[q]);
                            adj_words[ clique[q] ].push_back(clique[p]);
                        }
                    }
                }
            }
        }
    }

    return adj_words;
}
```

8.5 Summary to avoid performance issues

8.5.1 AVL trees

- The heights of left and right subtress differ at most by 1; not too deep.
- Operations such as `insert` must restore the tree.

8.5.2 Splay trees

- Nodes can get arbitrarily deep but after every access the tree is adjusted either using *zig-zag* or *zig-zig*.
- The net effect is that any sequence of M operations takes $O(MlogN)$ which the is the same as a balanced tree.

8.5.3 B-trees

- Balanced M-way (as opposed to binary trees) which are well suited the external memory model.

9 Lecture 09: Hashing

If keys are small integers, we can use an array to implement an unordered symbol table (known as **hash table**), by interpreting the key as an array index so that we can store the value associated with key i in array entry i, ready for immediate access. We can consider **hashing** (i.e., the implementation of a hash table) an extension of this simple method that handles more complicated types of keys. We reference key-value pairs using arrays by doing arithmetic operations to transform keys into array indices.

Hashing is a technique used for performing insertions, deletions, and finds in constant average time. Tree operations that require any ordering information among the elements are not supported efficiently.

Search algorithms that use hashing consist of two separate parts:

- Computing a **hash function** that transforms the search key into an array index. Ideally, different keys would map to different indices.
- Resolving collisions via a **collision-resolution process** that deals with situations where two or more different keys may hash to the same array index. Two different approaches to collision resolution are of relative importance: separate chaining and linear probing.

9.1 Hash functions

If we have an array that can hold M key-value pairs, then we need a hash function that can transform any given key into an index into that array. In other words, an integer in the range $[0, M-1]$.

We seek a hash function that is both *easy to compute* and *uniformly distributes the keys*: *for each key, every integer between* 0 *and* $M-1$ *should be equally likely (independently for every key).*

9.1.1 Hash function 1

```
int hash( const std::string& key, int tableSize ) {
    int hashVal = 0;

    for (char ch :  key) {
        hashVal += ch;
    }

    return hashVal % tableSize;
}
```

This hash function is simple to implement and computes an answer quickly. However if the table size is large, the function doesn't distribute the keys well. For instance, suppose that the table size is $10,007$. Suppose all the keys are 8 or fewer characters long (e.g., banana). Given that an ASCII character has an integer value between 0 and 127, the hash function typically can only values between 0 and $1,016$, which is 127×8.

9.1.2 Hash function 2

```
int hash( const std::string& key, int tableSize ) {
    return (key[0] + 27 * key[1] + 27 * 27 * key[2]) % tableSize;
}
```

This function assumes that `key` has at least three characters. The value 27 represents the number of letters in the English alphabet, plus a whitespace. This function examines ony the first three characters, but if these are random and the table is $10,007$, then we'd expect a reasonably equitable distribution. Unfornately, English is not random. Although there are $26^3 = 17,576$ possible combination of three characters (ignoring blanks), a check of a reasonably large dictionary reveals that the number of different legal combinations is actually only $2,851$. Although easily computable, this function isn't appropriate if the hash table is reasonably large.

9.1.3 Hash function 3

```
unsigned int hash( const std::string& key, int tableSize ) {
    unsigned int hashVal = 0;

    for (char ch :  key) {
        hashVal = 37 * hashVal + ch;
    }

    return hashVal % tableSize;
}
```

This hash function involves all characters in the `key` and can generally be expected to distribute well since it computes

$$\sum_{i=0}^{keySize-1} key[keySize - i - 1] \cdot 37^i$$

and brings the result into the proper range. The code computes a polynomial function (of 37) by using the Horner's rule. Another way of computing $h_k = k_0 + 37k_1 + 37^2k_2$ is by the formula $h_k = (k_2 \times 37 + k_1) \times 37 + k_0$.

Although not necessarily good at distributing the keys, the function is extremely simple and is reasonably fast. If the keys are very long, the hash function will take too long to compute. A common practice is not to use all the key's characters and letting the length and properties of the key influence the choice. For instance, the keys might've multiple parts, such as a mailing address, and the function could take a couple of characters from the stree address and perhaps a couple of characters from the city name and ZIP code.

9.2 Collision resolution

If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it. As mentioned above, there are several methods for dealing with this. However, the most simple ones are *separate chaining* and *open addressing*.

9.2.1 Closed addressing (Separate chaining)

This approach of resolving collisions is to build, for each of the M array indices, a linked list of the key-value pairs whose keys hash to that index. This method is known as *separate chaining* because items that collide are chained together in separate linked lists.

The basic idea is to choose M to be sufficiently large that the lists are sufficiently short to enable efficient search through a two-step process:

1. Hash to find the list that could contain the key.
2. Sequentially search through that list for the key.

For this section, we assume the keys to be the first 10 perfect squares $(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)$ and that the hashing function is simply `hash(x)= x mod 10`. For instance, `hash(81)` computes 1 as the array's index; so does `hash(1)`. Thus, they're chained together.

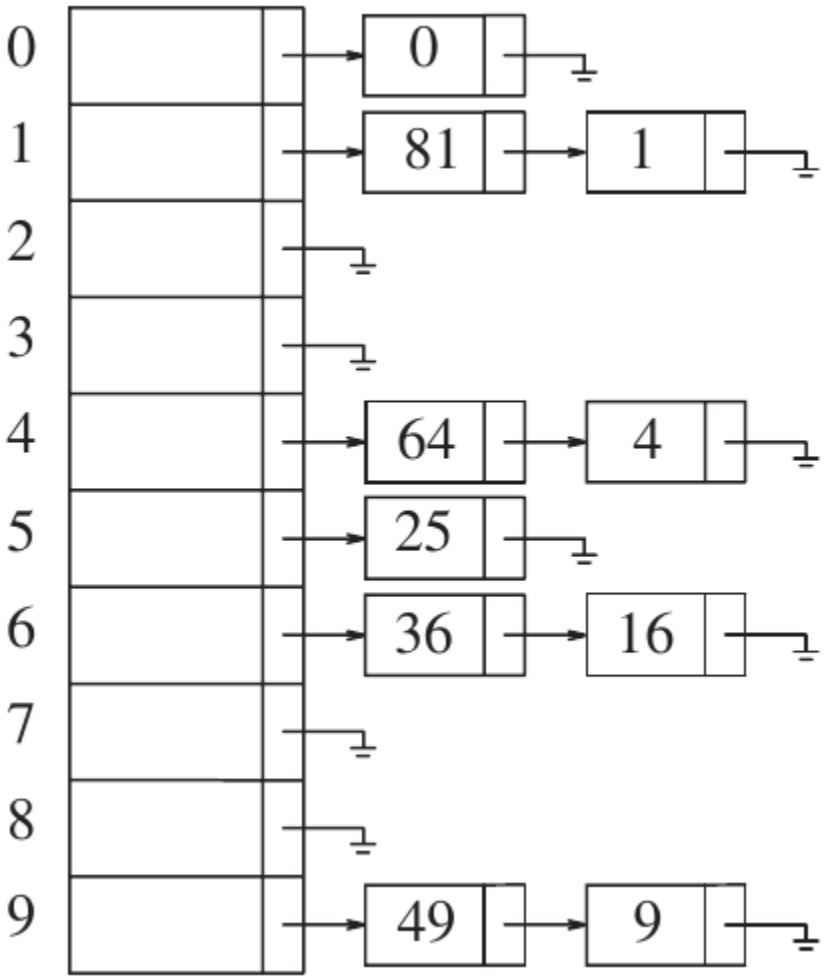


Figure 52: Separate chaining hash table

Implementation: implementations/HashTable.h

Class interface for separate chaining hash table

```
template<typename HashedObj>
class HashTable {
    public:
        explicit HashTable( int size = 0 );

        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>> _lists; // array of lists
        int _current_size;

        void _rehash();
        size_t _myhash( const HashedObj& x ) const;
}
```

The hash tables represented by the `HashTable` class works only for objects that provide a hash function and equality operators.

Hash functions Instead of requiring hash functions that take both the object and the table size as parameters, we have our hash functions take only the object as the parameter and return an appropriate integral type.

Search To perform a `search`, we use the hash function to determine which list to traverse. We then search the appropriate list.

Insertion To perform an `insert`, we check the appropriate list to see whether the element is already in place (if duplicates are expected, an extra data member is usually kept, and this data member would be incremented in the event of a match). If the element turns out to be new, it can be inserted at the front of the list, since it's convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

```
bool insert( const HashedObj& x ) {
    auto& list = _lists[ myhash(x) ];

    if (find(list.begin(),list.end(), x) != list.end()) {
       return false;
    }

    list.push_back(x);

    // rehash
    if (++_current_size > _lists.size()) {
        _rehash();
    }

    return true;
}
```

Separate chaining analysis First let's start by defining the load factor λ as the ration of the number of elements in the hash table to the table size. The average length of a list is λ. The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list.

In an unsuccessful search (i.e., miss), the number of nodes to examine is λ on average. Thus, its average worst-case is $O(1)+\lambda$. A successful search (i.e., hit) requires that about $1+(\lambda/2)$ links to be traversed.

This analysis shows that the table size is not really important but the load factor is. The general rule for separate chaining hashing is to make the table size about as large as the number of elements expected (in other words, let $\lambda \approx 1$).

General rules for separate chaining

- Make the table size as large as the number of expected elements.
- In code, $\lambda > 1$, we expand the table size by calling a function to **rehash**.
- It's a good idea to keep the table size prome to ensure good distribution of the keys.

10 Lecture 10: Hashing (cont.)

10.1 Collision resolution

10.1.1 Open addressing

Another approach to implementing hashing is to store N key-value pairs in a hash table of size $M > N$, relying on empty entries in the table to help with collision resolution. Such methods are called *open-*

addressing hashing methods.

This approach still uses linked lists, however unlike separate chaining, it tries alternative cells until an empty cell is found. More formally, cells $h_0(x), h_1(x), h_2(x), ...$ are tried in succession, where $h_i(x) = (hash(x) + f(i)) mod TableSize$, with $f(0) = 0$. The function, f, is the collision resolution strategy. Because all the data go inside the table, a bigger table is needed in such a scheme than for separate chaining hashing. Generally, the load factor should be below $\lambda = 0.5$ for a hash table that doesn't use separate chaining.

The simplest open-addressing method is called *linear probing*.

Linear probing When there is a collision (when we hash to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index). Linear probing is characterized by identifying three possible outcomes:

- Key equal to search key (i.e., search hit)
- Empty position (i.e., search miss)
- Key not equal to search key (i.e., try next entry)

We hash the key to a table index, check whether the search key matches the key there, and continue (incrementing the index, wrapping back to the beginning of the table if we reach the end) until finding either the search key or an empty table entry. It is customary to refer to the operation of determining whether or not a given table entry holds an item whose key is equal to the search key as a **probe**.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Hash table with linear probing, after each insertion

Figure 53: Linear probing

As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect, known as **primary clustering**, means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

The expected number of probes using linear probing is roughly $\frac{1}{2}(1 + 1/(1 - \lambda)^2)$ for insertions and unsuccessful searches, and $\frac{1}{2}(1 + 1/(1 - \lambda))$.

Random collision resolution: Assume a huge table where clustering is not an issue and each probe is indepedent of the previous probes, then the expected number of probes in miss equals the expected number of probes to fin an empty cell, that is, $1/(1 - \lambda)$.

Proof: The probability of selecting an empty cell equals $1 - \lambda = p$ (i.e., probability of success). The probability of selecting a non-empty cell is $\lambda = 1 - p$ (i.e., probability of failure). Finding an empty cell is like flipping a coin N times until success (assume coin is biaded having probability p of selecting success). The number of probes is thus a random variable X having a Geometric Probability Distribution. The expected value of X is thus $1/\text{Prob. of success} = 1/(1 - \lambda)$.

Quadratic probing Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect – the collision function is quadratic. The popular choice is $f(i) = i^2$.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.13 Hash table with quadratic probing, after each insertion

Figure 54: Quadratic probing

- `insert(89)`
  - $h_0(89) = ((89 + 0^2) \bmod 10) = 9$. Cell 9 unoccupied so insert 89.
- `insert(18)`
  - $h_0(18) = ((18 + 0^2) \bmod 10) = 8$. Cell 8 unoccupied so insert 18.
- `insert(49)`
  - $h_0(49) = ((49 + 0^2) \bmod 10) = 9$. Cell 9 occupied.
  - $h_1(49) = ((49 + 1^2) \bmod 10) = 0$. Cell 0 unoccupied so insert 49.
- `insert(58)`
  - $h_0(58) = ((58 + 0^2) \bmod 10) = 8$. Cell 8 occupied.
  - $h_1(58) = ((58 + 1^2) \bmod 10) = 9$. Cell 9 occupied.
  - $h_2(58) = ((58 + 2^2) \bmod 10) = 2$. Cell 2 unoccupied so insert 58.
- `insert(69)`
  - $h_0(69) = ((69 + 0^2) \bmod 10) = 9$. Cell 9 occupied.
  - $h_1(69) = ((69 + 1^2) \bmod 10) = 0$. Cell 0 occupied.
  - $h_2(69) = ((69 + 2^2) \bmod 10) = 3$. Cell 3 unoccupied so insert 69.

For linear probing, it is a bad idea to let the hash table get nearly full, because per- formance degrades. For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.

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 least half empty.

Proof: Let T be the table size and odd prime greater than 3. We show that the first $\lceil T/2 \rceil$ alternative locations (including the initial location $h_0(x)$) are all distinct. Two of these locations are $h(x) + i^2 (\text{mod } T)$ and $h(x) + j^2 (\text{mod } T)$, where $0 \leq i$ and $j \leq \lfloor T/2 \rfloor$. Suppose, for the sake of contradiction, that these locations are the same, but $i \neq j$. Then

$$
\begin{aligned}
h(x) + i^2 &= h(x) + j^2 \pmod T \\
i^2 &= j^2 \pmod T \\
i^2 - j^2 &= 0 \pmod T \\
(i-j)(i+j) &= 0 \pmod T
\end{aligned}
$$

Since T is prime, it follows that either $(i-j)$ or $(i+j)$ is equal to 0 (mod T). Since i and j are distinct, the first option is not possible. Since $0 \leq i$ and $j \leq \lfloor T/2 \rfloor$, the second option is also impossible. Thus, the first $\lceil T/2 \rceil$ alternative locations are distinct. If at most $\lfloor T/2 \rfloor$ positions are taken, then an empty spot can always be found.

10.2 Applications

When $log(N)$ is too big…

- Symbol tables in interpreters/compilers
- Real-time databases (in main memory or disk)
  - Air traffic control
  - Packet routing
  - Graphs where nodes are strings
  - Password checking
- Spell-checkers

When associative memory is needed…

- Dynamic programming
  - Cache results of previous computation
  - Chess endgames
- Text processing applications

10.3 Hash tables summary

- Used to index large amounts of data.
- Array index of each key is calculated using the key itself.

- Collisions between keys that hash to the same index are resolved using some collision resolution strategy (e.g., closed addressing, open addressing, etc.).
- Insertion, deletion and search occur at constant time.
- Hashing is widely used in database indexing, compilers, caching, password authentication, etc.

11 Lecture 11: Hashing (cont.)

11.1 Double hashing

For the collision resolution method known as **double hashing**, one popular choice is $f(i) = i \cdot hash_2(x)$ which says that we apply a second hash function to x and probe at a distance $hash_2(x)$, 2hash_{2}(x), …,$ and so on. A poor choice of $hash_2(x)$ would be disastrous. A few things to ensure:

- The function must never evaluate to zero. It is also important to make sure
- All cells can be probed. A function such as $hash_2(x) = R - (x mod R)$, with R a prime smaller than the table size T, will work well.

If the table size is not prime, it's possible to run out of alternative locations prematurely. However if double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy which makes double hashing theoretically interesting.

Nevertheless, *quadratic probing does not require the use of a second hash function* and is thus likely to be simpler and faster in practice, especially for keys like strings whose hash functions are expensive to compute.

11.2 Rehashing

If the table gets too full:

- The running time for the operations will start taking too long.
- Insertions might fail for open addressing hashing with quadratic resolution. This can happen if there are too many removals intermixed with insertions.

Solution: Build another table that is about twice as big (e.g., 70% full) (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.

11.2.1 Example

Suppose the elements $13, 15, 24$, and 6 are inserted into a linear probing hash table of size 7. The hash function is $h(x) = x$ mod 7 where x is the key (e.g., $h(13) = 6$, $h(15) = 1$, etc.). The resulting table is:

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 55: Hash table with linear probing with input $13, 15, 6, 24$

If 23 is inserted into the table, the resulting table will be over 70 percent full:

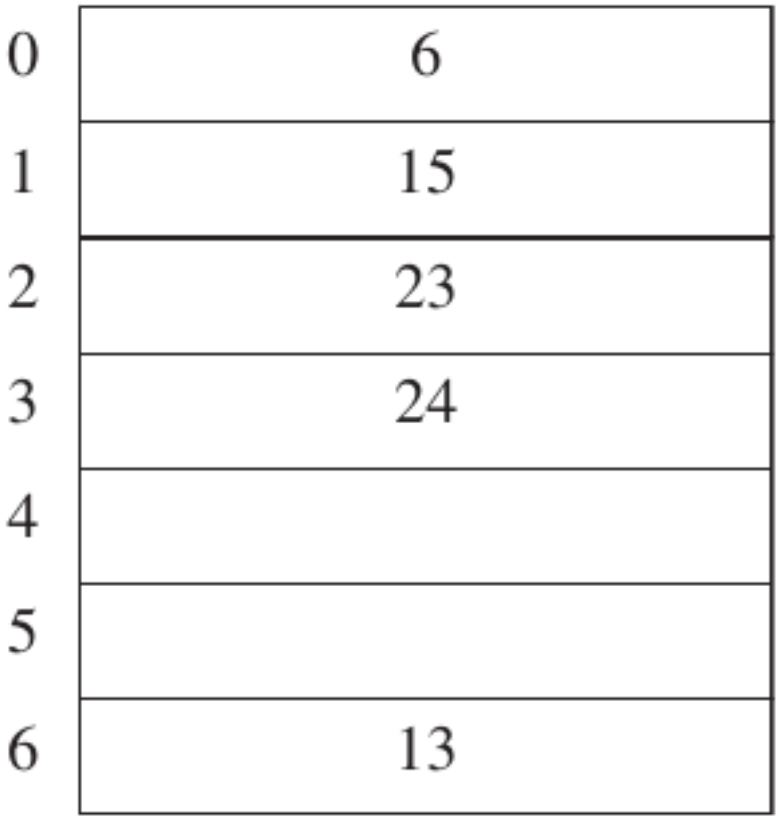


Figure 56: Hash table with linear probing after 23 is inserted

Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime that is twice as large as the old table size. The new hash function is then $h(x) = x \bmod 17$. The old table is scanned, and elements $6, 15, 23, 24$, and 13 are inserted into the new table:

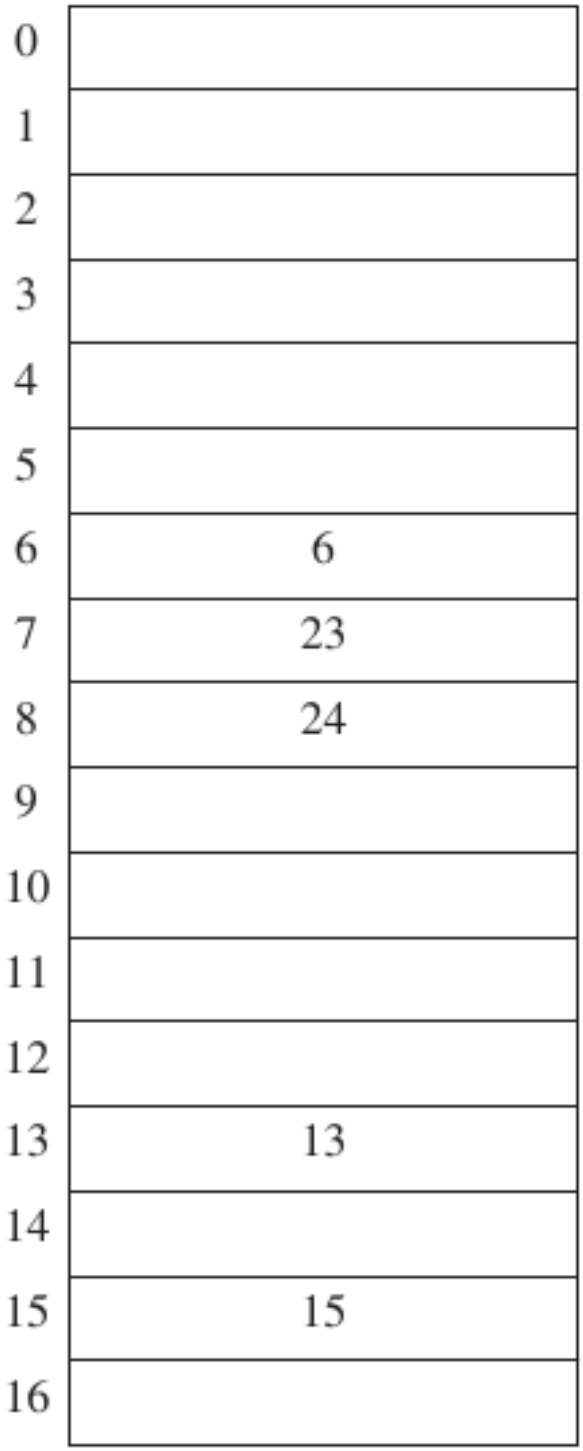


Figure 57: Hash table after reshahing

Rehashing can be implemented in several ways with quadratic probing. Some alternatives are:

- Rehash as soon as the table is half full.
- Rehash only when an insertion fails.
- A third, middle-of-the-road strategy is to rehash when the table reaches a certain load factor. Since performance does degrade as the load factor increases, the third strategy, implemented with a good cutoff, could be best.

Rehashing for separate chaining hash tables is similar.

11.3 STL's `unordered_set` and `unordered_map`

- Hash table implementation of sets and maps.
- Same functionality as `set` and `map`, but no ordered capabilities:
  - Items in `unordered_set` (or keys in `unordered_map`) must provide an overloaded `operator==` and a hash function.
  - `unordered_set` and `unordered_map` can be instantiated with function objects that provide a hash function and equality operator.

In the word-changing example (Lecture 8), there were three maps:

1. A map in which the key is a *word length*, and the value is a collection of all words of that word length. Because the order in which word lengths are processed does not matter, the map an `unordered_map`.
2. A map in which the key is a *representative*, and the value is a collection of all words with that representative. Because the representatives are not even needed after the second map is built, the map can be an `unordered_map`.
3. A map in which the key is a *word*, and the value is a collection of all words that differ in only one character from that word. This map can also be an `unordered_map`.

The performance of an `unordered_map` can often be superior to a map, but it is hard to know for sure without writing the code both ways.

11.4 Hash Tables with Worst-Case O(1) Access

For separate chaining, assuming a *load factor* of 1, this is one version of the classic **balls and bins problem**:

> Given N balls placed randomly (uniformly) in N bins, what is the expected number of balls in the most occupied bin?

The answer is well known to be $\Theta(logN/loglogN)$, meaning that on average, we expect some queries to take nearly logarithmic time. Similar types of bounds are observed (or provable) for the length of the longest expected probe sequence in a probing hash table.

We would like to obtain $O(1)$ worst-case cost. In some applications, such as hardware implementations of lookup tables for routers and memory caches, it is especially important that the search have a definite (i.e., constant) amount of completion time. Let us assume that N is known in advance, so no rehashing is needed. If we are allowed to rearrange items as they are inserted, then $O(1)$ worst-case cost is achievable for searches.

11.5 Extendible hashing

Extendible hashing deals with the cases where the amount of data is too large (i.e., the hash table is huge) to fit in main memory. We assume that at any point we have N records to store; the value N changes over time. Furthermore, at most M records fit in one disk block and $M < N$.

With either probing hashing or separate chaining hashing, we have the following problems:

- collisions could cause several blocks to be examined during a search, even for a well-distributed hash table.
- when the table gets too full, an extremely expensive rehashing step must be performed, which requires $O(N)$ disk accesses.

The alternative solution is **extendible hashing** which allows a search to be performed in two disk accesses. Insertions also require few disk accesses.

11.5.1 B-tree approach for extendible hashing

Recall that a B-tree has depth $O(log_{M/2}N)$; as M increases, the depth of a B-tree decreases. In theory, we could M to sufficiently large for the depth of the B-tree to become 1. Then any search after the first would take one disk access, since presumably the root node could be stored in main memory. However, the problem is that the *branching factor* is so high that it would take considerable processing to determine the leaf in which the data is in. By reducing the time to perform this step, a practical scheme can be reached; this is the strategy extendible hashing uses.

Let's suppose our data consists of several 6-bit integers:

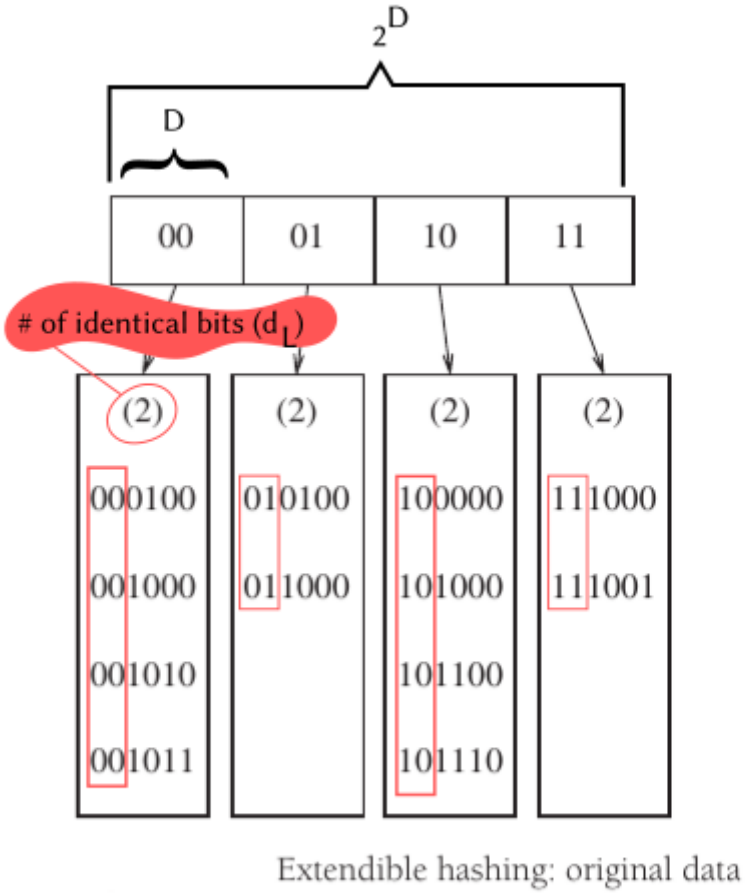


Figure 58: Extendible hashing, original data

The root of the tree contains 4 pointers determined by the leading two bits of the data. Each leaf has up to $M = 4$ elements; in each leaf the first two bits are identical for all integers. We can let D be the number of bits used by the root, usually known as the **directory**. The number of entries in the directory is thus 2^D. For a leaf L, d_L is the number of leading bits that are common to all elements for the given leaf; d_L will depend on the particular leaf, and $d_L \leq D$.

Suppose we want to insert the key 100100. This would be placed in the third leaf (since its two leading bits are 10), but the third leaf is already full. We thus split this third leaf into two leaves, which are now determined by the first $d_L = 3$ bits for these two leaves. This requires increasing the directory size to 3 and the number of entries in the directory is now $2^3 = 8$:

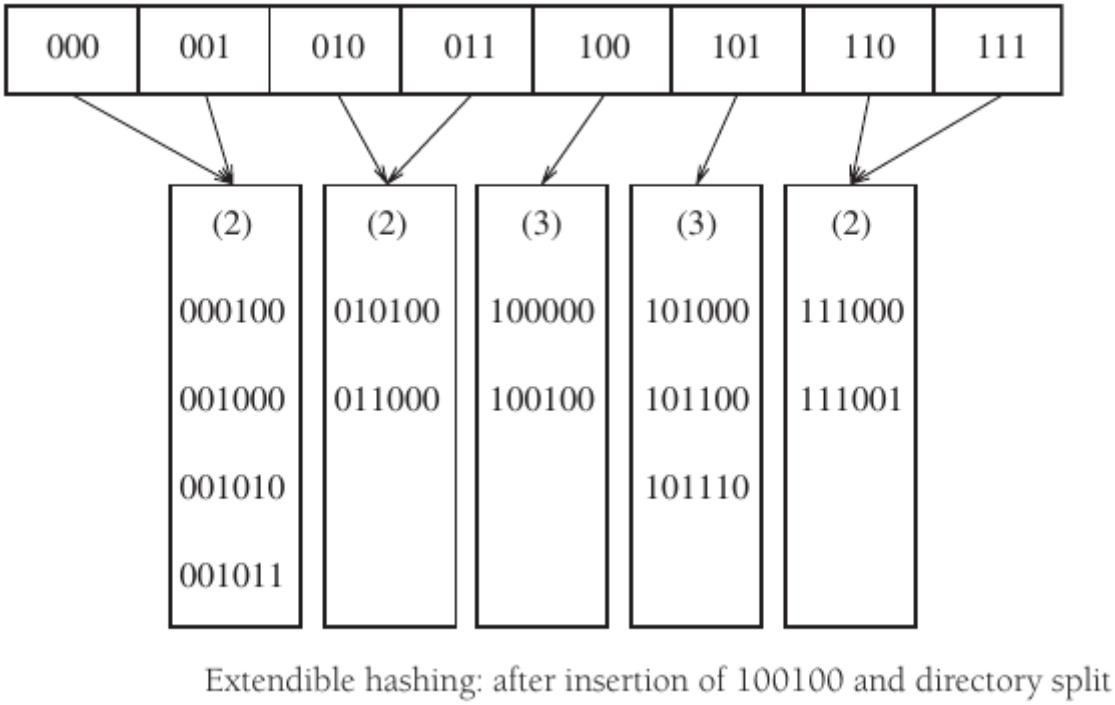


Figure 59: Extendible hashing, insertion and directory split

Notice that all leaves not involved in the split are now pointed to by two adjacent directory entries (e.g., first leaf is pointed to by 000 and 001). Thus, although an entire directory is rewritten, none of the other leaves are accessed.

Now suppose the key 000000 is inserted. Then the first leaf is split, generating two leaves with $d_L = 3$. Since $D = 3$, the only change required in the directory is the updating of the 000 and 001 pointers:

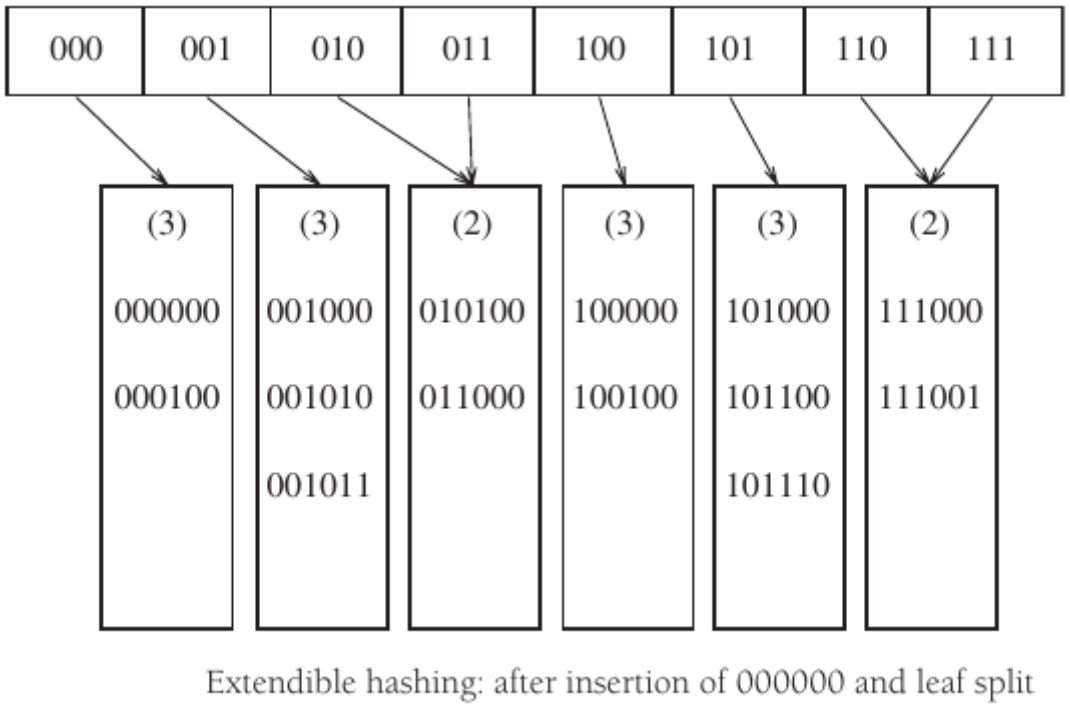


Figure 60: Extendible hashing, insertion and leaf split

All in all, this simple strategy provides quick access times for `insert` and search operations on large databases.

There are a few important details to consider:

1. It's possible that several directory splits will be required if the elements in a leaf agree in more than $D + 1$ leading bits.
2. There's the possibility of duplicate keys; if there are more than M duplicates, then this algorithm doesn't work at all.

These possibilities suggest that it is important for the bits to be fairly random. This can be accomplished by hashing the keys into a reasonably long integer—hence the name of *extendible* hashing.

11.5.2 Perfomance properties

These properties are derived after a very difficult analysis. These results are based on the reasonable assumption that the bit patterns are uniformly distributed.

- The expected number of leaves is $(N/M)log_2e$. Thus the average leaf is $ln2 = 0.69$ full. This is the same as for B-trees, which is not entirely surprising,since for both data structures new nodes are created when the $(M + 1)$th entry is added.
- The expected size of the directory (i.e., 2^D) is $O(N^{1+1/M}/M)$. If M is very small, then the directory can get unduly lage. In this case, we can have the leaves contain pointers to the records instead of the actual records, thus increasing the value of M. This adds a second disk access to each search operation in order to maintain a smaller directory. If the directory is too large to fit in main memory, the second disk access would be needed anyway.

11.6 Hashing summary

- Constant average for insert/search operations.
- Load factor λ is crucial:
  - roughly 1 for separate chaining
  - less than 0.5 for probing
  - Can change with rehashing (an expensive operation)
- BSTs could also be used
- If no ordering is required, hash table `set`/`map` is probably better, although both must be tried to decide which is better.

12 Heaps

Priority queues are a special kind of binary tree called a **heap**, which means "a disorganized pile." This is in contrast to binary search trees that can be thought of as a highly organized pile.

12.1 Model

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

- `insert`, for inserting an element into the priority queue; and
- `deleteMin`, which finds, returns, and removes the minimum element in the priority queue.

The `insert` operation is the equivalent of `enqueue`, and `deleteMin` is the priority queue equivalent of the queue's `dequeue` operation.

12.2 Simple implementations

There are several obvious ways to implement a priority queue:

- Use a simple linked list, performing insertions at the front in $O(1)$ and traversing the list, which requires $O(N)$ time, to delete the minimum.
- Use an always-sorted linked list sorted; this makes insertions expensive ($O(N)$) and deleteMins cheap ($O(1)$).
- Use a binary search tree which gives an $O(log\ N)$ average running time for both operations. This is true in spite of the fact that although the insertions are random, the deletions are not. Using a search tree could be overkill because it supports a host of operations that are not required. The basic data structure we will use will not require links and will support both operations in $O(log\ N)$ worst-case time. Insertion will actually take constant time on average, and our implementation will allow building a priority queue of N items in linear time, if no deletions intervene.

If choosing between a simpled linked list or an always-sorted linked list, the former is probably the better idea of the two, based on the fact that there are never more deleteMins than insertions.

12.3 Binary heap: an implicit binary tree

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 is easy to show that a complete binary tree of height h has between $2h$ and $2h + 1 - 1$ nodes which implies that the height of a complete binary tree is $log\ N$, clearly $O(log\ N)$.

Eytzinger's method allows us to represent a complete binary tree as an array by laying out the nodes of the tree in breadth-first order In this way, the root is stored at position 1, the root's left child is stored at position 2, the root's right child at position 3, the left child of the left child of the root is stored at position 4, and so on.

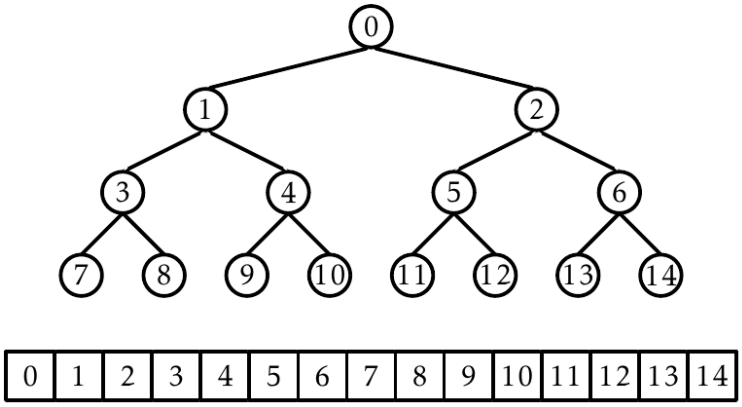


Figure 61: Eytzinger's method represents a complete binary tree as an array

If we apply Eytzinger's method to a sufficiently large tree, some patterns emerge:

- The left child of the node at index i is at index $2i + 1$.
- The right child of the node at index i is at index $2i + 2$.
- The parent of the node at index i is at index $\frac{i-1}{2} = \lfloor i/2 \rfloor$.

For a heap, links aren't required and the operations required to traverse the tree are extremely simple and likely to be very fast on most computers. A heap data structure will, then, consist of an array and an integer representing the current heap size.

12.3.1 Heap-order property

A binary heap uses the above mentioned technique to implicitly represent a complete binary tree in which the elements are *heap-ordered* (*heap-order property*):

> In a heap, the value stored at any index i is not smaller than the value stored at index parent(i), with the exception of the root value, which has no parent. It follows that the smallest value in the priority queue is therefore stored at position 0 (the root).

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.

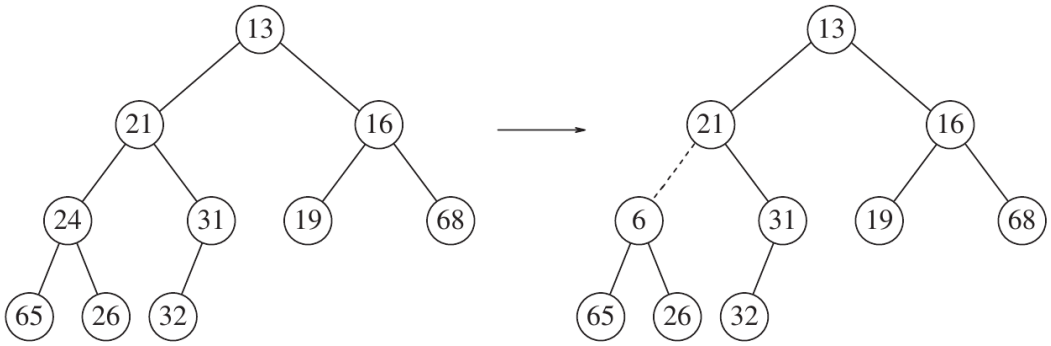


Figure 62: Two completes (only the left tree is a heap)

12.3.2 Basic operations

implementation: implementations/BinaryHeap

`insert` To insert an element X into the heap, we create a hole in the next available location, since otherwise, the tree will not be complete. If X can be placed in the hole without violating heap order, then we do so and are done. Otherwise, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until X can be placed in the hole.

This general strategy is known as a **percolate up**; the new element is percolated up the heap until the correct location is found.

```
Input: X is the element to insert
* array is the underlying structure for the heap.
* currentSize is the number of elements in the array.

insert( X ):
    if currentSize = (array.size - 1):
        array.resize(array.size * 2)

    # percolate up
    Int hole = currentSize++
    copy = X

    array[0] = copy
    while x < array[hold div 2]:
        array[hole] = array[hole div 2]
        hole = hole div 2
    array[hole] = array[0]
```

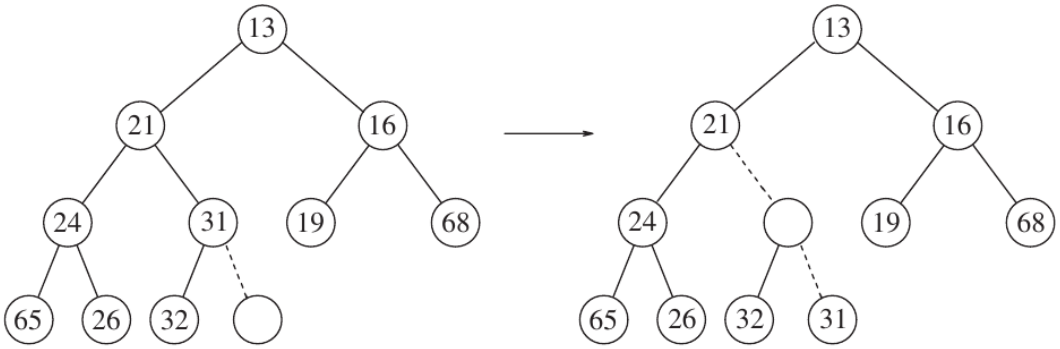


Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

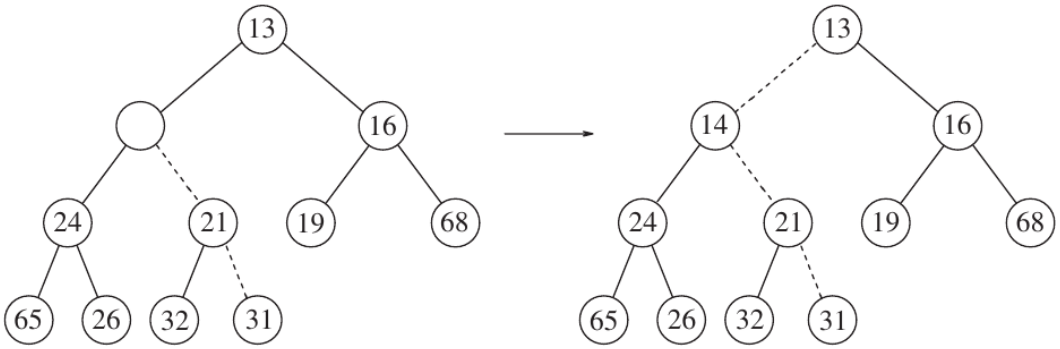


Figure 6.7 The remaining two steps to insert 14 in previous heap

Figure 63: Inserting an element in a binary heap

`deleteMin` In this case, finding the minimum is easy; the hard part is removing it. When the minimum is removed, a hole is created at the root (i.e., the smallest element is in the root). Since the heap now becomes one smaller, it follows that the last element X in the heap must move somewhere in the heap. If X can be placed in the hole, then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until X can be placed in the hole. Thus, our action is to place X in its correct spot along a path from the root containing *minimum* children. This general strategy is known as a **percolate down**.

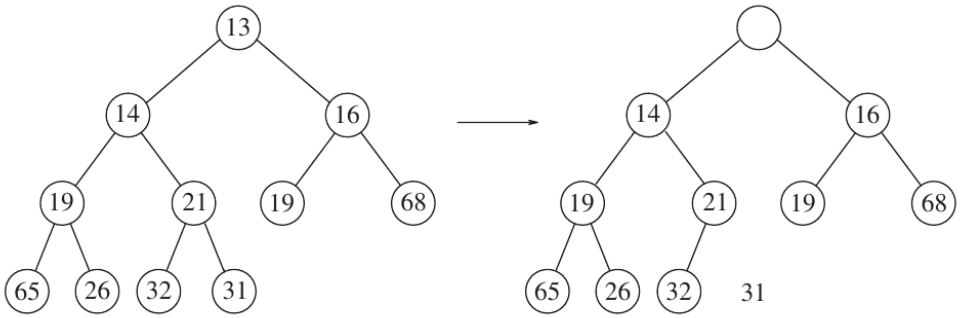


Figure 64: Creation of the hole at the root

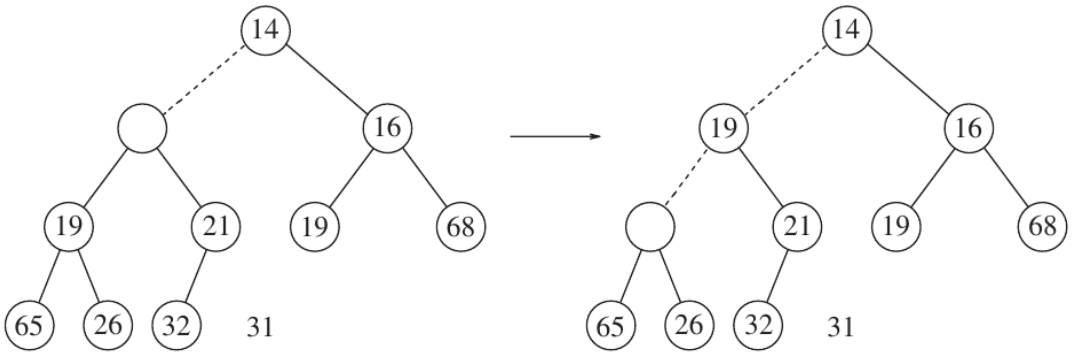


Figure 65: Next two steps in deleteMin

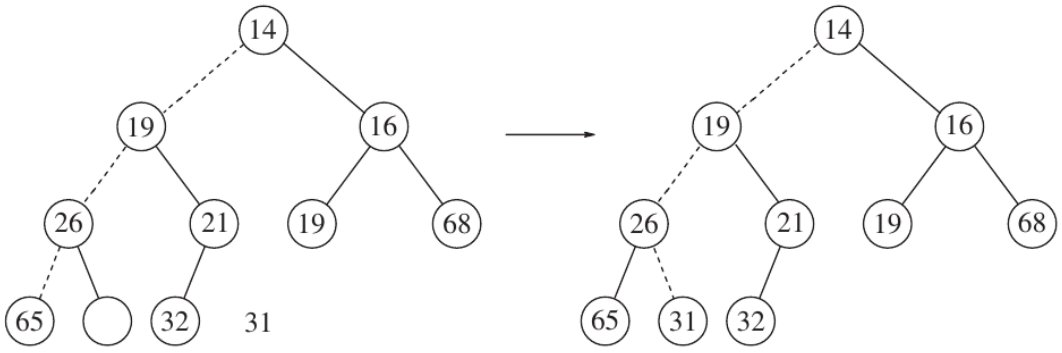


Figure 66: Last two steps in deleteMin

A frequent implementation error in heaps occurs when there are an even number of elements in the heap, and the one node that has only one child is encountered. You must make sure not to assume that there are always two children, so this usually involves an extra test.

One extremely tricky solution is always to ensure that your algorithm thinks every node has two children. Do this by placing a sentinel, of value higher than any in the heap, at the spot after the heap ends, at the start of each *percolate down* when the heap size is even. You should think very carefully before attempting this, and you must put in a prominent comment if you do use this technique. Although this eliminates the need to test for the presence of a right child, you cannot eliminate the requirement that you test when you reach the bottom, because this would require a sentinel for every leaf.

The worst-case running time for this operation is $O(log\ N)$. On average, the element that is placed at the root is percolated almost to the bottom of the heap (which is the level it came from), so the average running time is $O(log\ N)$.

12.3.3 Other operations

decreaseKey

- The `decreaseKey(p, delta)` operation lowers the value of the item at position `p` by a positive amount `delta`.
- It might violate the heap order, it must be fixed by a *percolate up*.
- It could be useful to system administrators: They can make their programs run with highest priority.

increaseKey

- The `increaseKey(p, delta)` operation increases the value of the item at position `p` by a positive amount `delta`.
- This is done with *percolate down*.
- Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

remove

- The `remove(p)` operation removes the node at position `p` from the heap.
- This is done by performing `decreaseKey(p, Inf)` and then performing `deleteMin()`.
- When a process is terminated by a user (instead of finishing normally), it must be removed from the priority queue.

buildHeap The binary heap is sometimes constructed from an initial collection of items. This constructor takes as input N items and places them into a heap. Obviously, this can be done with N successive `insert`s. Since each insert will take $O(1)$ average- and $O(log\ N)$ worst-case time, the total running time of this algorithm would be $O(N)$ average- (from $N \times O(1)$) but $O(N\ log\ N)$ (from $N \times O(log\ N)$) worst-case. Since this is a special instruction and there are no other operations intervening, and we already know that the instruction can be performed in linear average time, it is reasonable to expect that with reasonable care a linear time bound can be guaranteed.

The general algorithm is to place the N items into the tree in any order, maintaining the structure property. Then, if `percolateDown(i)` percolates down from node `i`, the `buildHeap` can be used by the constructor to create a heap-ordered tree.

```
BinaryHeap( const std::vector<T>& items )
    : array{ items.size() + 10 }, currentSize{ items.size() }
{
    for (int i = 0; i < items.size(); i++) {
        array[i + 1] = items[i];
    }
    buildHeap()
}

/*
Establish heap order property from an arbitrary
arrangement of items. Runs in linear time.
*/
void buildHeap() {
    for (int i = currentSize / 2; i > 0; i--) {
       percolateDown(i);
    }
}

/*
Percolate down in the heap.
The hole is the index at which the percolate begins.
*/
void percolateDown( int hole ) {
    int child;
    T tmp = std::move(array[hole]);

    while (hold * 2 <= currentSize) {
        child = hole * 2;
        if (child != current && array[child + 1] < array[child]) child++;
        if (array[child] < tmp) array[hole] = std::move(array[child]);
        else break;
        hold = child;
    }

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

To bound the running time of `buildHeap`, we must bound the sum of the heights of all the nodes in the heap. What we would like to show is that this sum is $O(N)$.

Theorem: For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

Proof: The tree consists of $2^0 = 1$ node at height h, $2^1 = 2$ nodes at height $h - 1$, 2^2 nodes at height $h - 2$, and in general 2^i nodes at height $h - i$. The sum of the heights of all the nodes is then

$$\begin{aligned} S &= \sum_{i=0}^{h} 2^i (h - i) \\ &= h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \cdots + 2^{h-1} \end{aligned}$$

Multiplying by 2 gives the equation:

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \cdots + 2^h$$

Subtracting these equations:

$$2S - S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1)$$

which proves the theorem.

A complete tree is not a perfect binary tree, but the result we have obtained is an upper bound on the sum of the heights of the nodes in a complete tree. Since a complete tree has between 2^h and 2^{h+1} nodes, this theorem implies that this sum is $O(N)$, where N is the number of nodes.

13 Heaps (cont.)

13.1 The selection problem

> Suppose you have a group of N numbers and would like to determine the k^{th} largest. The input is a list of N elements, which can be totally ordered, and an integer k.

There are several algorithms to solve the problem:

- **1st algorithm:** Read the elements into an array and sort them, and return the appropriate element. Assuming a simple sorting algorithm, the running time is $O(N^2)$.
- **2nd algorithm:** Read k elements into an array and sort them. The smallest of these is in the k^{th} position. We process the remaining elements one by one. As an element arrives, it is compared with the k^{th} element in the array. If it's larger, then the k^{th} element is removed, and the new element is placed in the correct place among the remaining $k-1$ elements. When the algorithm ends, the element in the k^{th} position is the answer. The running time is $O(N \cdot k)$.
- **3rd algorithm:** Assume we're interested on finding the k^{th} smallest element. Read the N elements into an array. We then apply the `buildHeap` algorithm to this array. Finally, we perform k `deleteMin` operations. The last element extracted from the heap is the answer.

  The worst-case timing is $O(N)$ to construct the heap, if `buildHeap` is used, and $O(log\ N)$ for each `deleteMin`. Since there are k `deleteMin`s, we obtain a total running time of $O(N + klog\ N)$. If $k = O(N/log\ N)$, then the running time is dominated by the `buildHeap` operation and is $O(N)$. For larger values of k, the running time is $O(klog\ N)$. If $k = \lceil N/2 \rceil$, then the running time is $\Theta(Nlog\ N)$.

  Notice that if we run this program for $k = N$ and record the values as they leave the heap, we will have essentially sorted the input file in $O(Nlog\ N)$ time.
- **4th algorithm:** Here we return to the original problem and find the k^{th} largest element. We use the same idea from the 2nd algorithm. At any point in time we will maintain a set S of the k largest elements. After the first k elements are read, when a new element is read it is compared with the k^{th} largest element, which we denoted by S_k. Notice that S_k is the the smallest element in S. If the new element is larger, then it replaces S_k in S. S will then have a new smallest, which may or may not not be the newly added element. At the end of the input, we find the smallest element in S and return it as the answer.

  The first k elements are placed into the heap in total $O(k)$ time with a call to `buildHeap`. The time process each of the remaining elements $O(1)$, to test if the element goes into S, plus $O(log\ k)$, to delete S_k and insert the new element if this is necessary. Thus, the total time is $O(k + (N-k)log\ k) = O(Nlog\ k)$. This algorithm also gives a abound of $\Theta(Nlog\ N$ for finding the median.

13.2 d-heaps

Binary heaps are so simple that they are almost always used when priority queues are needed. A **d-heap** is a simple generalization of a heap where all nodes all nodes have d children (thus, a binary heap is a 2-heap).

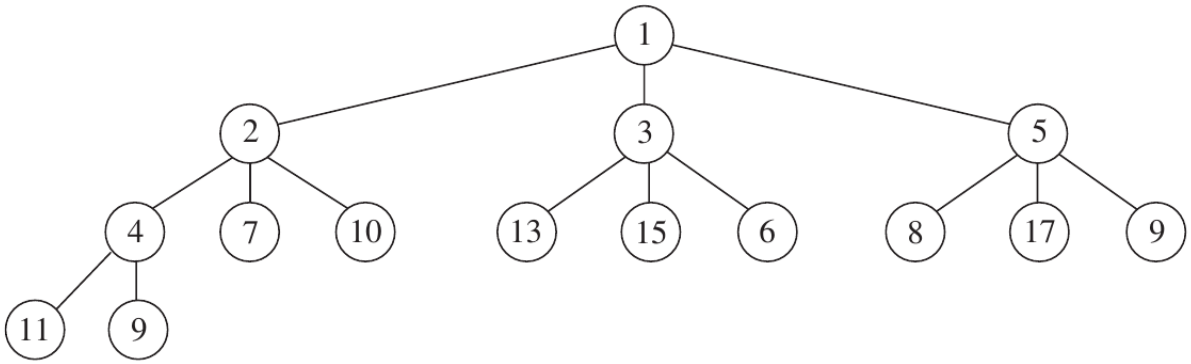


Figure 67: A d-heap (d = 3)

A d-heap is much shallower than a binary heap, which improves the running time of `insert`s to $O(log_d\ N)$. However, for large d, the `deleteMin` operation is more expensive, because even though the tree is shallower, the minimu of d children must be found, which takes $d - 1$ comparisons using a standard algorithm. This raises the time for this operation to $O(dlog_dN)$. If d is a constant, both running times are, of course, $O(log\ N)$.

d-heaps are interesting in theory, because there are many algorithms where the number of insertions is much greater than the number of `deleteMin`s (and thus a theoretical speedup is possible). They are also of interest when the priority queue is too large to fit entirely in main memory. In this case, a d-heap can be advantageous in much the same way as B-trees. Finally, there is evidence suggesting that 4-heaps may outperform binary heaps in practice.

The most glaring weakness of the heap implementation, aside from the inability to perform `find`s, is merging two heaps into one. This extra operation is known as a `merge`. There are quite a few ways to implement heaps so that the running time of a merge is $O(log\ N)$.

13.3 Leftist heaps

HERE

13.3.1 Leftist heap property

We define the **null path length**, $npl(X)$, of any node X to be the length of the shortest path from X to a node without two children. Thus, the npl of a node with zero or one child is 0, while $npl(nullptr) = -1$.

13.3.2 Leftist heap property

13.4 Skew heaps

A **skew heap** is a self-adjusting version of a leftist heap that is incredibly simple to implement. The relationship of skew heaps to leftist heaps is analogous to the relation between splay trees and AVL trees. Skew heaps are binary trees with heap order, but there is no structural constraint on these trees. Unlike leftist heaps, no information is maintained about the null path length of any node. The right path of a skew heap can be arbitrar- ily long at any time, so the worst-case running time of all operations is O(N). However, as with splay trees, it can be shown (see Chapter 11) that for any M consecutive operations, the total worst-case running time is O(M log N). Thus, skew heaps have O(log N) amortized cost per operation.

14 Priority Queues(Binomial Queues) and Sorting

14.1 Binomial queues

Both leftist and skew heaps support merging, insertion, and `deleteMin` in $O(logN)$ time per operation. Binomial queues support all three operations in $O(logN)$ worst-case time per operation, **but** insertions take constant time on average.

Structurally, a **binomial queue** differs from all the priority queue implementations in that a *binomial queue is not a heap-ordered tree but rather a collection of heap-ordered trees*, known as a **forest**. Each of the heap-ordered trees is of a constrained form known as a binomial tree. There is at most one binomial tree of every height. A binomial tree of height 0 is a one-node tree; a binomial tree, B_k , of height k is formed by attaching a binomial tree, B_{k-1} , to the root of another binomial tree, B_{k-1}.

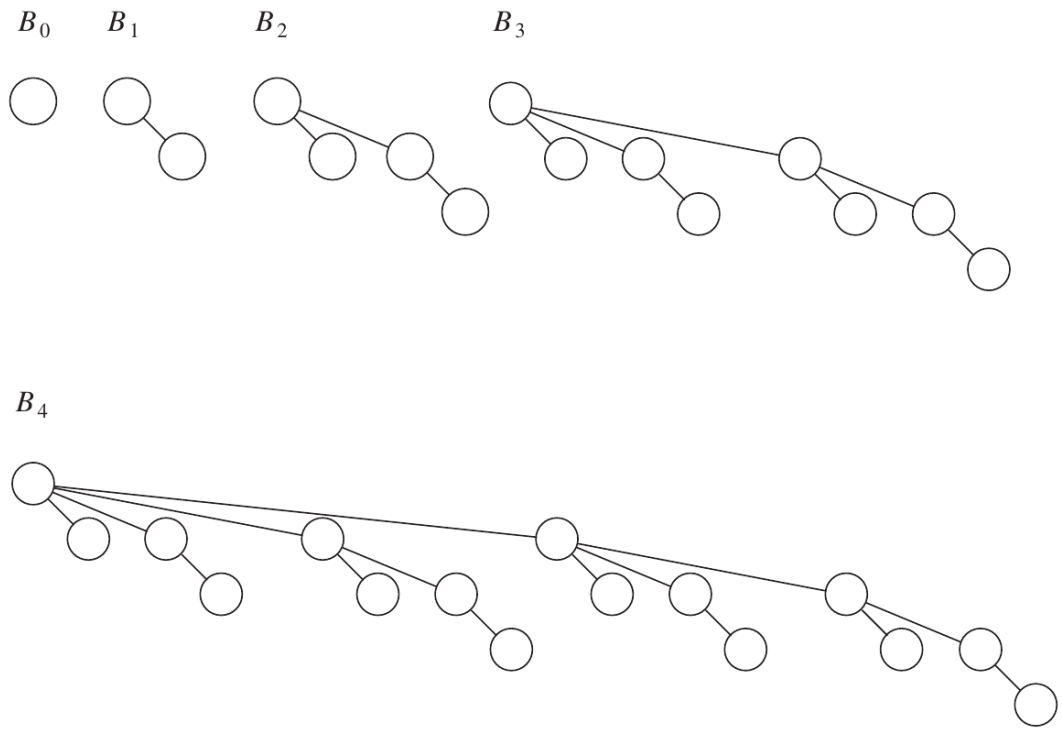


Figure 68: Binomial trees B_0, B_1, B_2, B_3 and B_4

A binomial tree, B_k, consists of a root with children $B_0, B_1, ..., B_{k-1}$. Binomial trees of height k have exactly 2^k nodes, and the **number of nodes at depth** d is the **binomial coefficient** $\binom{k}{d}$.

If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can represent a priority queue of any size by a collection of binomial trees. Suppose, we've a priority queue of size 13. This could be represented by the forest B_3, B_2, B_0. In binary representation, $13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$, which also represents the fact that B_3, B_2, and B_0 are present in the representation and B_1 is not (i.e., 0×2^1).

For example, a priority queue of 6 elements could be represented as the binomial queue down below.

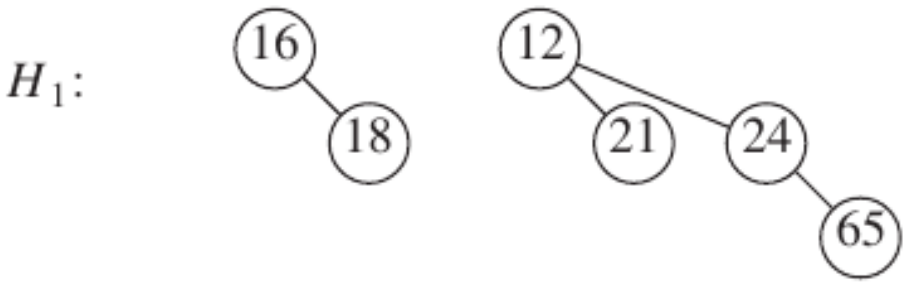


Figure 69: Binomial queue H_1 with six elements

14.1.1 Operations

Merge Merging two binomial queues is a conceptually easy operation. Consider the two binomial queues, H_1 and H_2 with 6 and 7 elements respectively down below.

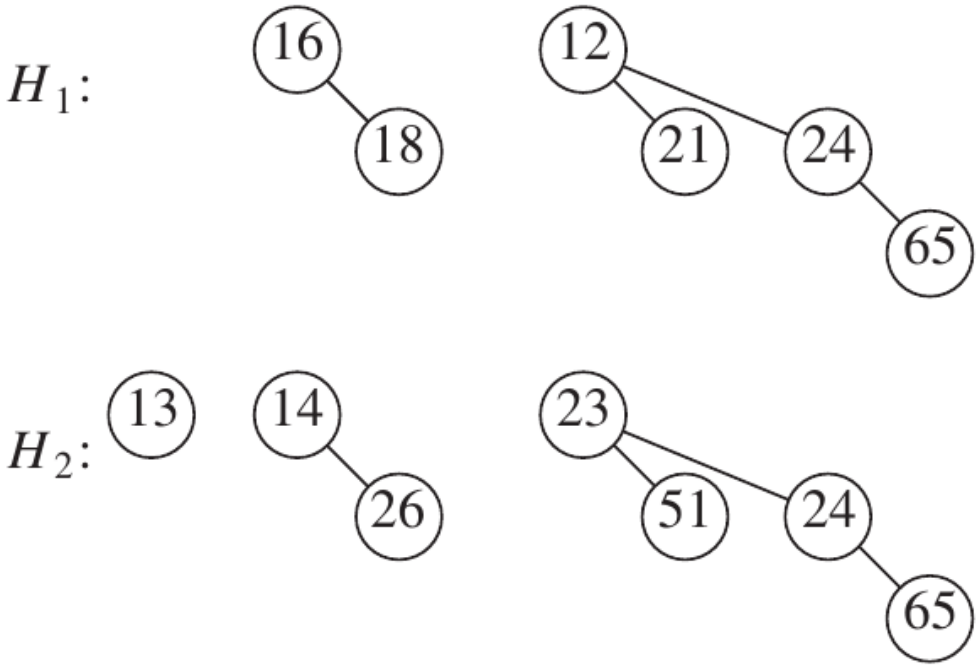


Figure 70: Two binomial queues H_1 and H_2

The merge is performed by essentially adding the two queues together and performed as follows:

- Let H_3 be the new binomial queue. Since H_1 has no binomial tree of height 0 and H_2 does, we can just use the binomial tree of height 0 in H_2 as part of H_3. Next, we add binomial trees of height 1. Since both H_1 and H_2 have binomial trees of height 1, we merge them by making the larger root a subtree of the smaller, creating a binomial tree of height 2, as shown down below. Thus, H_3 will not have a binomial tree of height 1.

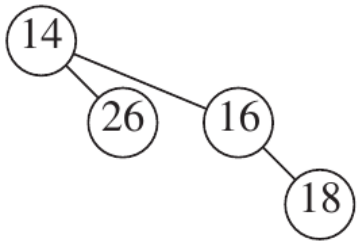


Figure 71: Merge of the two B_1 trees in H_1 and H_2

- There are now three binomial trees of height 2, namely, the original trees of H_1 and H_2 plus the tree formed by the previous step. We keep one binomial tree of height 2 in H_3 and merge the other two, creating a binomial tree of height 3. Since H_1 and H_2 have no trees of height 3, this tree becomes part of H_3 and we are finished. The resulting binomial queue is shown down below.

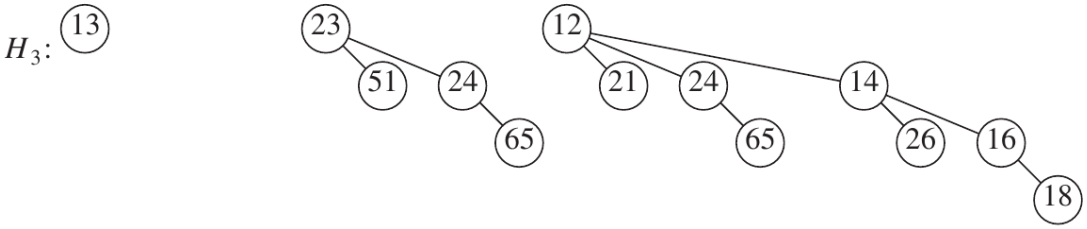


Figure 72: Binomial queues H_3: result of merging $H-1$ and H_2

Since merging two binomial trees takes constant time with almost any reasonable implementation, and there are $O(logN)$ binomial trees, the merge takes $O(logN)$ time in the worst case. To make this operation efficient, *we need to keep the trees in the binomial queue sorted by height*, which is certainly a simple thing to do.

Insertion Insertion is just a special case of merging, since we merely create a one-node tree and perform a merge. The worst-case time of this operation is likewise $O(logN)$. More precisely, if the

priority queue into which the element is being inserted has the property that the smallest nonexistent binomial tree is B_i, the running time is proportional to $i + 1$.

For example, H_3 (from previous examples) is missing a binomial tree of height 1, so the insertion will terminate in two steps. Since each tree in a binomial queue is present with probability $1/2$, it follows that we expect an insertion to terminate in two steps, so the average time is constant.

As an example, let's see the binomial queues that are formed by inserting the numbers 1 through 7 in order.

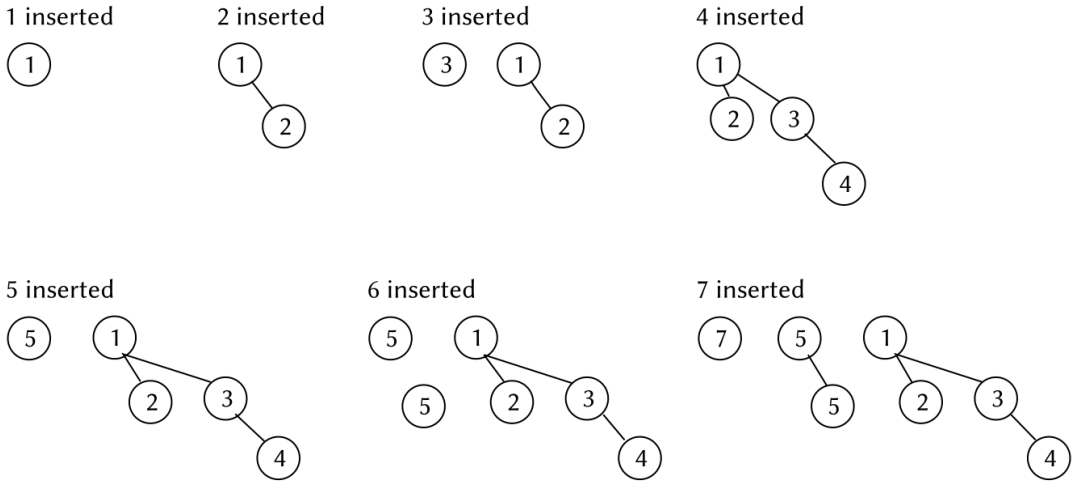


Figure 73: Insertions into binomial queue

While inserting, we find inserting 4 is a bad case. We merge 4 with B_0, ontaining a new tree of height 1. We then merge this tree with B_1, obtaining a tree of height 2, which is the new priority queue. We count this as 3 steps (two merges plus the stopping case). The next insertion after 7 is inserted is another bad case and would require 3 tree merges.

deleteMin A `deleteMin` can be performed by first finding the binomial tree with the smallest root. Let this tree be B_k, and let the original priority queue be H. We remove the binomial tree B_k from the forest of trees in H, forming the new binomial queue H'. We also remove the root of B_k, creating binomial trees B_0, B_1, ..., B_{k-1}, which collectively form priority queue H''. We finish the operation by merging H' and H''.

As an example, suppose we perform a `deleteMin` on H_3 (from previous examples), The minimum root is 12, so we obtain the two priority queues H' and H'' as shown below.

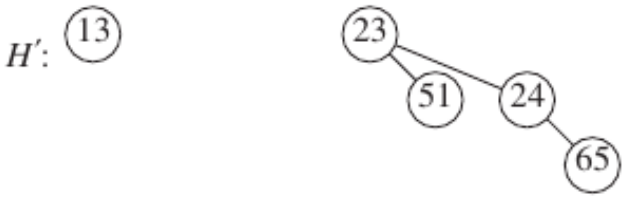


Figure 74: Binomial queue H', containing all the binomial trees in H_3 except B_3

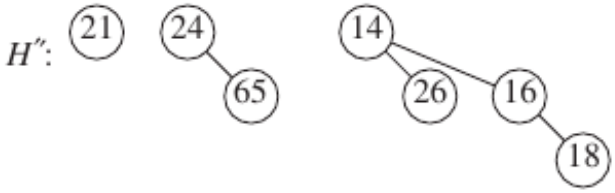


Figure 75: Binomial queue H'': B_3 with 12 removed

The binomial queue that results from merging H' and H'' is the final answer, which is shown down below.

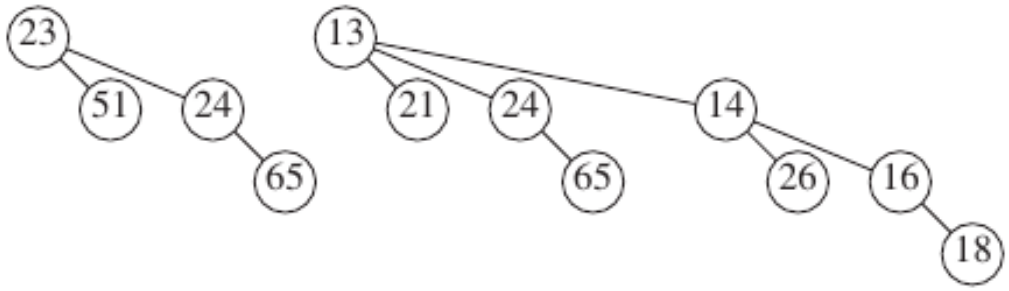


Figure 76: Result of applying `deleteMin` to H_3

14.1.2 Implementation

deleteMin The `deleteMin` operation requires the ability to find all the subtrees of the root quickly, so the standard representation of general trees is required:

- The children of each node are kept in a linked list, and each node has a pointer to its first child (if any). This operation also requires that the children be ordered by the size of their subtrees.
- We also need to make sure that it is easy to merge two trees. When two trees are merged, one of the trees is added as a child to the other. Since this new tree will be the largest subtree, it makes sense to maintain the subtrees in decreasing sizes. Only then will we be able to merge two binomial trees, and thus two binomial queues, efficiently. The binomial queue will be an array of binomial trees.

To summarize, then, each node in a binomial tree will contain the data, first child, and right sibling. The children in a binomial tree are arranged in decreasing rank.

The binomial queue depicted in the figure below

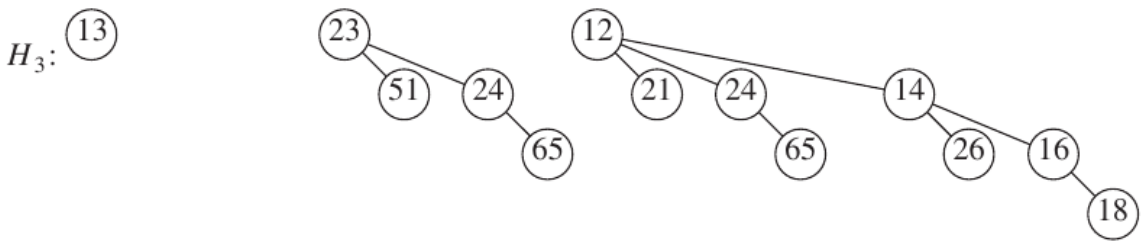


Figure 77: Binomial queue H_3 drawn as a forest

can be represented as a linked list as shown in the figure.

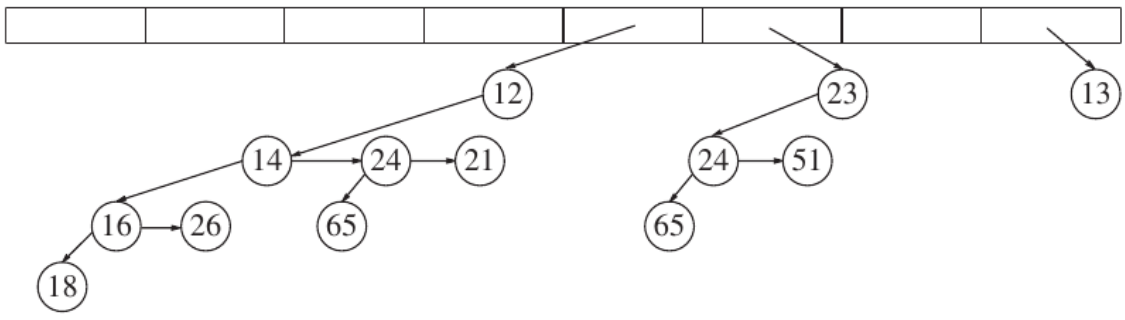


Figure 78: Representation of binomial queue H_3

14.1.3 Summary

Basic binary heap

- $O(logn)$ for insertion and `deleteMin`
- $O(1)$ average for insertion
- $O(n)$ to build the heap

Heaps with $O(logn)$ merge operation

- Leftist heap – Simple recursive data structure
  - Does not have $O(1)$ average for insertion
- Skew heap – Amortized version of leftist heap
- Binomial queue – Based on a forest. Simple to describe and has the same complexity guarantees as the binary heap.

14.2 Sorting

14.2.1 Comparison-based sorting

In **comparison-based sorting**, the following conditions are assumed:

- Each algorithm is passed an array containing the elements; we assume all array positions contain data to be sorted. We will assume that N is the number of elements passed to our sorting routines.
- We will also assume the existence of the $<$ and $>$ operators, which can be used to place a consistent ordering on the input. Besides the assignment operator, these are the only operations allowed on the input data.

This interface is unlike the STL sorting algorithms, where sorting is accomplished by use of the function template `sort`. The parameters to sort represent the start and endmarker of a (range in a) container and an optional comparator:

```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator cmp );
```

The iterators must support random access. The sort algorithm does not guarantee that equal items retain their original order (if that is important, use `stable_sort` instead of `sort`).

14.2.2 Insertion order

Insertion sort is one of the simplest algorithms and it consists of $N-1$ **passes**. For pass $p=1$ through $N-1$, insertion sort ensures that the elements in positions 0 through p are in sorted order. Insertion

sort makes use of the fact that elements in positions 0 through $p-1$ are already known to be in sorted order.

Original	34	8	64	51	32	21	Positions Moved
After $p=1$	8	34	64	51	32	21	1
After $p=2$	8	34	64	51	32	21	0
After $p=3$	8	34	51	64	32	21	1
After $p=4$	8	32	34	51	64	21	3
After $p=5$	8	21	32	34	51	64	4

Figure 79: Insertion sort after each pass

```
template<typename Comparable>
void insertionSort( std::vector<Comparable>& a ) {
    for (int p = 1; p < a.size(); p++) {
        Comparable tmp = std::move(a[p]);

        int j;
        for (j = p; j > 0 && tmp < a[j-1]; j--) {
            a[j] = std::move(a[j-1]);
        }
        a[j] = std::move(tmp);
    }
}
```

In a STL implementation of insertion sort, the sort routines receive a pair of iterators that represent the start and endmarker of a range. A two-parameter sort routine uses just that pair of iterators and presumes that the items can be ordered, while a three-parameter sort routine has a function object as a third parameter.

```
/*
The two-parameter version calls the three-parameter version, * using C++11
decltype.
*/
template<typename Iterator>
void insertionSort( const Iterator& begin, const Iterator& end ) {
    insertionSort(begin, end, less<decltype(*begin)>{});
}
```

```
template<typename Iterator, typename Comparator>
void insertionSort( const Iterator& begin, const Iterator& end, Comparator
    lessThan) {
    if (begin == end) { return };

    Iterator j;
    for (Iterator p = begin + 1; p != end; p++) {
        auto tmp = std::move(*p);
        for (j = pl; j !+ begin && lessThan(tmp, *(j-1)); j--) {
            *j = std::move(*(j-1));
        }
        *j = std::move(tmp);
    }
}
```

The worst-case running time for the insertion sort algorithm is $\Theta(N^2)$ and its best-case running time (when the input is presorted) is $O(N)$, in which case a single comparison per item is performed.

14.2.3 A lower bound for simple sorting algorithms

An **inversion** in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$. Take for example the input list $34, 8, 64, 51, 32, 21$. This list has 9 inversions:

```
(34, 8)
(34, 32)
(34, 21)
(64, 51)
(64, 32)
(64, 21)
(51, 32)
(51, 21)
(32, 21)
```

Notice that this is exactly *the number of swaps* that needed to be (implicitly) performed by insertion sort. This is always the case, because

- swapping two adjacent elements that are out of place removes exactly one inversion, and
- a sorted array has no inversions

Since there is $O(N)$ other work involved in the algorithm, the running time of insertion sort is $O(I+N)$, where I is the number of inversions in the original array. Thus, insertion sort runs in linear time if the number of inversions is $O(N)$.

Theorem: The average number of inversions in an array of N distinct elements is $\frac{N(N-1)}{4})$.

> **Proof:**
>
> For any list, L, of elements, consider L_r, the list in reverse order. The reverse list of the example is 21, 32, 51, 64, 8, 34. Consider any pair of two elements in the list (x, y) with $y > x$. Clearly, in exactly one of L and L_r this ordered pair represents an inversion. The total number of these pairs in a list L and its reverse L_r is $\frac{N(N-1)}{2}$. Thus, an average list has half this amount, or $\frac{N(N-1)}{4}$ inversions.

This theorem implies that insertion sort is quadratic on average. It also provides a very strong lower bound about any algorithm that only exchanges adjacent elements.

Theorem: Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N2)$ time on average.

> **Proof:**
>
> The average number of inversions is initially $\frac{N(N-1)}{4} = \Omega(N^2)$. Each swap removes only one inversion, so $\Omega(N^2)$ swaps are required.

15 Lecture 15: Sorting (cont.)

15.1 Shellsort

- Named after the its inventor, Donald Shell.
- Among the first algorithm to break the quadratic time barrier.
- It works by comparing elements that are distant from each other; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared.

Shellsort uses a sequence, $h_1, h_2, \dots, h_t$, called the **increment sequence**. Any increment sequence will do as long as $h_1 = 1$, but some choices are better than others. After a phase, using some increment h_k, for every i, we have $a[i] \leq a[i + h_k]$ (where this makes sense); all elements spaced h_k apart are sorted. The file is then said to be h_k-sorted.

The idea behind Shellsort is to arrange the list of elements so that, starting anywhere, considering every h^{th} element gives a sorted list. In general, for each position, i, in $h_k, h_k + 1, \dots, N - 1$, place the element in the correct spot among $i, i - h_k, i - 2h_k$, and so on. This action performs an insertion sort on h_k independent of subarrays; after all, Shellsort is a generalization of inserion sort.

A popular (but poor choice) for increment sequence is to use the sequence suggested by Shell: $h_t = \lfloor N/2 \rfloor$, and $h_k = \lfloor h_{k+1}/2 \rfloor$. The following C++ function implements Shellsort using this sequence:

```
template<typename Comparable>
void shellsort( std::vector<Comparable>& a ) {
    for (int gap = a.size() / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < a.size(); i++) {
            Comparable tmp = std::move(a[i]);
            int j = i;

            for (; j >= gap && tmp < [j-gap]; j -= gap) {
                a[j] = std::move(a[j-gap]);
            }
            a[j] = std::move(tmp);
        }
    }
}
```

Take for example the list $62, 83, 18, 53, 7, 17, 95, 86, 47, 69, 25, 28$. Using the above function, we obtain the following:

```
orig   => 62, 83, 18, 53, 7, 17, 95, 86, 47, 69, 25, 28
gap: 6 => 62, 83, 18, 53, 7, 17, 95, 86, 47, 69, 25, 28
gap: 3 => 53, 7, 17, 62, 25, 18, 69, 83, 28, 95, 86, 47
gap: 1 => 7, 17, 18, 25, 28, 47, 53, 62, 69, 83, 86, 95
```

15.1.1 Worst-case analysis

Theorem: The worst-case running of Shellsort using Shell's increments is $\Theta(N^2)$.

> **Proof:**
>
> Consider that N is a power of 2, which means all increments are even except for the last one which is 1. We place the $N/2$ largest numbers at even positions and the $N/2$ smallest numbers at odd positions. To finish the proof, we must show the upper bound of $O(N^2)$:
>
> - At pass h_k: A pass with increment h_k consists of h_k inserion sorts of about N/h_k elements. Since insertion sort is quadratic, the total cost of a pass is $O(h_k(N/h_k)^2) = O(N^2/h_k)$.
> - Total cost of all passes: $O(\Sigma_{i=1}^{t} N^2/h_i) = O(N^2\Sigma_{i=1}^{i} 1/h_i)$. Because the increments form a geometric series with common ration 2, and the largest term in the series is $h_1 = 1$, $\Sigma_{i=1}^{t} 1/h_i < 2$. Thus we obtain a total bound of $O(N^2)$.
>
> The problem with Shell's increments is that pairs of increments are not necessarily relatively prime, and thus the smaller increment can have little effect. Among better improvements over Shell's original sequence, there are:
>
> - Hibbard's increment sequence, which gives better results in practice (and theoretically). His

> increments are of the form $1, 3, 7, ..., 2^k - 1$. The key difference here is that consecutive increments have no common factors (i.e., they're prime numbers). The Worst-case running time is $\Theta(N^{3/2})$.
>
> - Sedgewick's increment sequence. This sequence is of the form $1, 5, 19, 41, ...$, in which the terms are either of the form $9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$. The worst-case running time is $O(N^{4/3})$.

15.2 Heapsort

- Uses binary heap implemented as array
- $O(NlogN)$ time bound for both worst- and average-case running time.
- In-place algorithm, uses O(1) extra memory.

The algorithm boils down to

1) starting with a given array,
2) build binary heap from N element, which is $O(N)$, and
3) perform N `deleteMax()` operations which gives $O(NlogN)$.

The C++ code for the Heapsort algorithm is as follows. Unlike the binary heap, where the data begin at array index 1, the array for heapsort contains data at position 0 so minor changes are required.

```
template<typename Comparable>
void heapsort( std::vector<Comparable>& a ) {
    // buildHeap
    for (int i = a.size()/2 - 1;   i >= 0; i--)
        percolateDown(a, i, a.size());

    for (int j = a.size() - 1; j>0; j--) {
        std::swap(a[0], a[j]);
        percolateDown(a, 0, j);
    }
}

// Returns the index of the left child.
inline int leftChild( int i ) { return 2 * i + 1; }

template<typename Comparable>
void percolateDown( std::vector<Comparable>& a , int i, int n )  {
    int child;
    Comparable tmp;

    for (tmp = std::move(a[i]); leftChild(i) < n; i = child) {
        child = leftChild(i);
        if (child != n - 1 && a[child] < a[child+1])
            child++;
        if (tmp < a[child])
            a[i] = std::move(a[child]);
        else
            break;
    }

    a[i] = std::move(tmp);
}
```

15.3 Mergesort

- Run in $O(NlogN)$ worst-case running time.
- Number of comparisons is near optimal.
- Recursive
- Fundamental operation is the merging of two sorted lists, which takes $O(N)$ and needs for extra O(N) space to merge arrays.
- Good for linked list since there's no need for random access.
- There's no need for extra space to merge sublists.
- Stable

```
template<typename Comparable>
void mergeSort( std::vector<Comparable>& a ) {
    std::vector<Comparable> tmpArray(a.size());

    mergeSortInternal(a, tmpArray, 0, a.size()-1);
}

template<typename Comparable>
void mergeSortInternal(
    std::vector<Comparable>& a,   // array to sort
    std::vector<Comparable>& tmp, // array to place merged result
    int left,                     // leftmost index of the subarray
    int right,                    // rightmost index of the subarray
) {

    if (left < right) {
        int center = (left + right) / 2;              // get middle index
        mergeSortInternal(a, tmp, left, center);      // sort left
            subarray
        mergeSortInternal(a, tmp, center + 1, right); // sort right
            subarray
        merge(a, tmp, left, center + 1, right);       // merge sorted
            subarrays
    }
}

template<typename Comparable>
void merge(
    std::vector<Comparable>& a,   // array of Comparable items
    std::vector<Comparable>& tmp, // array to place merged result
    int leftPos,                  // leftmost index of subarray
    int rightPos,                 // index of the start of the second half
    int rightEnd                  // rightmost index of subarray
) {

    int leftEnd = rightPos = 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while (leftPos <= leftEnd && rightPos <= rightEnd) {
        if (a[leftPos] <= a[rightPos])
            tmp[tmpPos++] = std::move(a[leftPos++]);
        else
            tmp[tmpPos++] = std::move(a[rightPos++]);
    }

    // copy rest of left half
    while (leftPos <= leftEnd)
        tmp[tmpPos++] = std::move(a[leftPos++]);

    // copy rest of right half
    while (rightPos <= rightEnd)
        tmp[tmpPos++] = std::move(a[rightPos++]);

    for (int i = 0; i < numElements; i++, rightEnd--)
        a[rightEnd] = std::move(tmp[rightEnd]);
}
```

15.3.1 Worst-case analysis

We have to write a recurrence relation for the running time. We will assume that N is a power of 2 so that we always split into even halves. For $N = 1$, the time to mergesort is constant, which we will denote by 1. Otherwise, the time to mergesort N numbers is equal to the time to do two recursive mergesorts of size $N/2$, plus the time to merge the two sorted subarrays (which is linear). Mathematically, this could expressed as follows:

$$
\begin{aligned}
T(1) &= 1 \\
T(N) &= 2T(N/2) + N
\end{aligned}
$$

This is a recurrence relation and finding Mergesort's running time devolves into solving it:

$$
\begin{aligned}
T(N) &= 2T(N/2) + N \\
T(N) &= 4T(N/4) + 2N \\
T(N) &= 8T(N/8) + 3N \\
&\dots \\
T(N) &= 2^k T(N/2^k) + k \cdot N
\end{aligned}
$$

Using $k = logN$, we obtain

$$
T(N) = N \cdot T(1) + N \cdot logN = N \cdot logN + N
$$

Thus, mergesort's running time is $O(N\log N)$.

16 Lecture 16: The disjoint sets class

The disjoint sets class is an efficient data structure to solve the equivalence problem which is easy to implement (e.g., routines read just a a few lines of code and simple array can be used), extremely fast (i.e., constant average time per operation), and very interesting from a theoretical point of view (e.g., its analysis is extremely difficult).

16.1 Equivalent relations

A **relation** R is defined on a set S if for every pair of element (a, b), where the elements a and b are elements of the set S, $a\ R\ b$ is either true or false. If $a\ R\ b$ is true, then a is related to b.

An **equivalence relation** is a relation R that satisfies three properties:

- (*Reflexive*) $a\ R\ a$, for all $a \in S$.
- (*Symmetric*) $a\ R\ b$ if and only if $b\ R\ a$.
- (*Transitive*) $a\ R\ a$ and $b\ R\ c$ implies that $a\ R\ c$.

16.2 Dynamic equivalence problem

Given an equivalence relation $\sim$, the natural problem is to decide, for any a and b, if $a \sim b$.

If the relation is stored as a two-dimensional array of Boolean variables, then, of course, this can be done in constant time. The problem is that the relation is usually not explicitly, but rather implicitly, defined.

Suppose the equivalence relation is defined over the five-element set $\{a_1, a_2, a_3, a_4, a_5\}$. There are 25 pairs of elements (e.g., $a_1 \sim a_2$, $a_1 \sim a_3$, etc), each of which is either related or not related. As an equivalence relation, we have:

- Reflexivity: $a_i \sim a_i$ where $i \in 1 \ldots 5$.
- Simmetry: $a_2 \sim a_1$, $a_3 \sim a_2$
- Transitivity: $a_1 \sim a_2$, $a_2 \sim a_3$, $a_1 \sim a_3$

The **equivalence class** of an element $a \in S$ is the subset of S that contains all the elements that are related to a. To decide if $a \sim b$, we need only to check whether a and b are in the same equivalence class, which provides the strategy to solve the equivalence problem.

16.3 Disjoint sets

- Input is initially a collection of N sets, each with one element.
- Initial representation is that all relations (except reflexive relations) are false.
- Each set has a different element, so that $S_i \cap S_j = \emptyset$, which makes the sets **disjoints** (i.e., no elements in common).

16.4 Union-find

Find:

- This operation reyurns the name of the set (i.e., equivalence class) containing a given element.

Union:

- If we want to add the relation $a \sim b$, then we must first find if a and b are related:
  - Perform a `find` operations on a and b and check if they're in the same equivalent class.
  - If already related, do nothing.
  - If not realted, apply the `union` to the equivalence classes of both `a` and `b`, and create a new class. The result of the `union` is to create a new set $S_k = S_i \cup S_j)$, which destroy the originals and preserve the **disjointness** of all sets.

In conjunction, this algorithm is known as the disjoint set **union-find** algorithm. This algorithm

- is **dynamic** because, during the course of the algorithm, the sets can change via the `union` operation.
- operates **online**, meaning that whenever `find` is performed, it must give an answer before continuing. Similar to an oral exam.
- can also operate **offline**, meaning the sequence of `union`s and `find`s is given, and you need to calculate final sets. Similar to a written exam.

16.5 List-based implementations

The list-based implementation of union/find:

- Each set is stored in a sequence represented with a *linked list*.
- Each node should store an object containing the element and a reference to the set name.

When doing a unoin, we always move elements from the smaller set to the larger one.Thus, each time an elment is moved it goes to a set whose size is at least double its old set. This means an element can be moved at most $O(logN)$ times.

Therefore, the total time needed to do M `find`s and $N-1$ `union`s is $O(M + NlogN)$.

16.5.1 A better implementation

Here:

- Each set (i.e., equivalence class) is represented by a tree, since each element in a tree has the same root.
- The root is used to name the set (i.e., the representative).

- Initially, each set contains a single element.
- Can be implemented using an array. For exaple, each entry $s[i]$ in the array represents the parent of element i. If i is a root, then $s[i] = -1$.

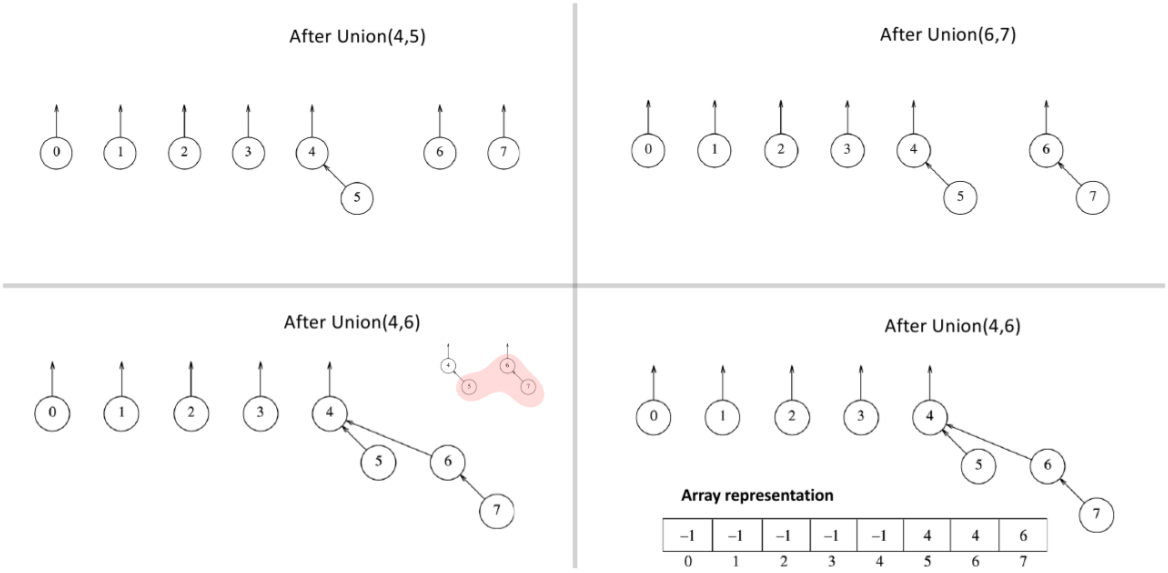


Figure 80: Example of multiple `union` operations

16.6 C++ implementation

```
class DisjointSets {
    public:
        /*
        Construct the disjoint sets object.
        */
        explicit DisjointSets( int numElements ) : s{ numElements - 1 } {

        }

        /*
        Return the set containing x.
        */
        int find( int x ) const {
            if (s[x] < 0) { return x; }
            else          { return find(s[x]); }

        }

        /*
        Get the union of two sets.

        Not the best way.
        */
        void union( int root1, root2 ) {
            s[root2] = root1;
        }

        private:
            std::vector<int> s;
}
```

16.7 Smart union algorithms

16.7.1 Union by size

The unions above were performed rather arbitrarily, by making the second tree a subtree of the first. A simple improvement is always to make the smaller tree a subtree of the larger, breaking ties by any method; we call this approach **union-by-size**.

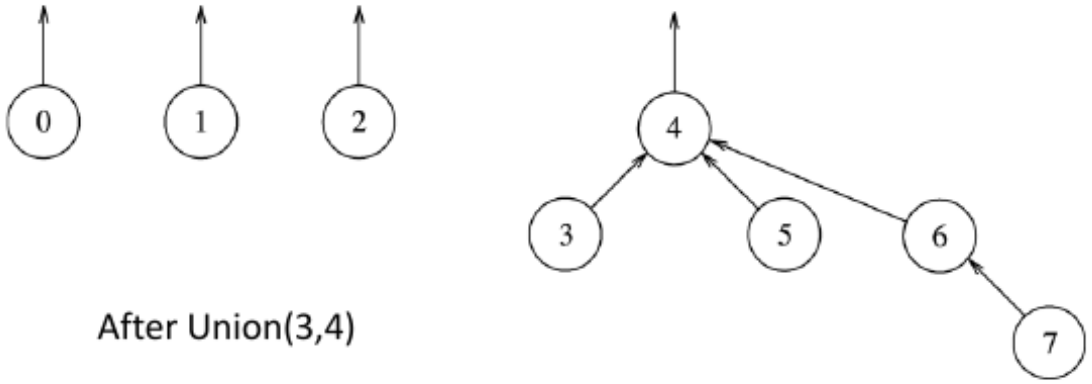


Figure 81: Union by size

We can prove that *if unions are done by size, the depth of any node is never more than* $logN$: To see this, note

- A node is initially at depth 0. When depth increases as a result of a union, it is placed in a tree that is at least twice as large as before.
- Thus, the node's depth can be increased at most $log\ N$ times. This implies that the running time for a find operation is $O(log\ N)$, and a sequence of M operations takes $O(M log N)$.

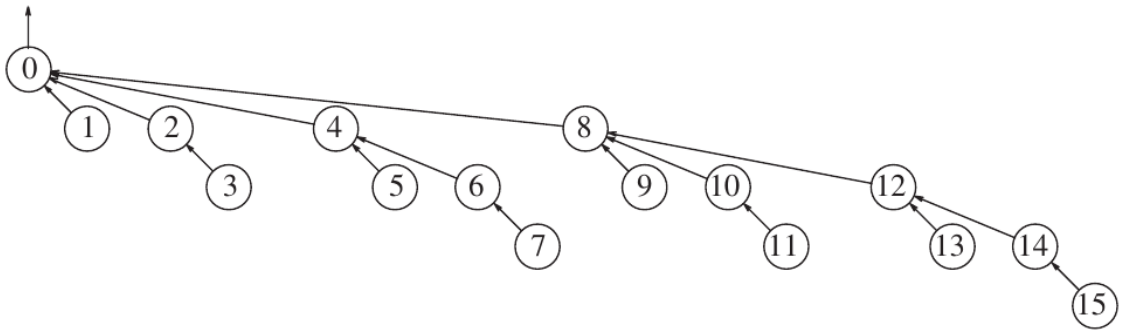


Figure 82: Worst-case tree for $N = 16$

16.7.2 Union by height

This alternative implementation, which also guarantees that all the trees will have depth at most $O(log\ N)$, is **union-by-height**:

- Keep track of the height, instead of the size, of each tree.

- Perform unions by making the shallow tree a subtree of the deeper tree.

This is an easy algorithm, since the height of a tree increases only when two equally deep trees are joined (and then the height goes up by one). Thus, *union-by-height is a trivial modification of union-by-size*. Since heights of 0 would not be negative, we actually store the negative of height, minus an additional 1. Initially, all entries are -1.

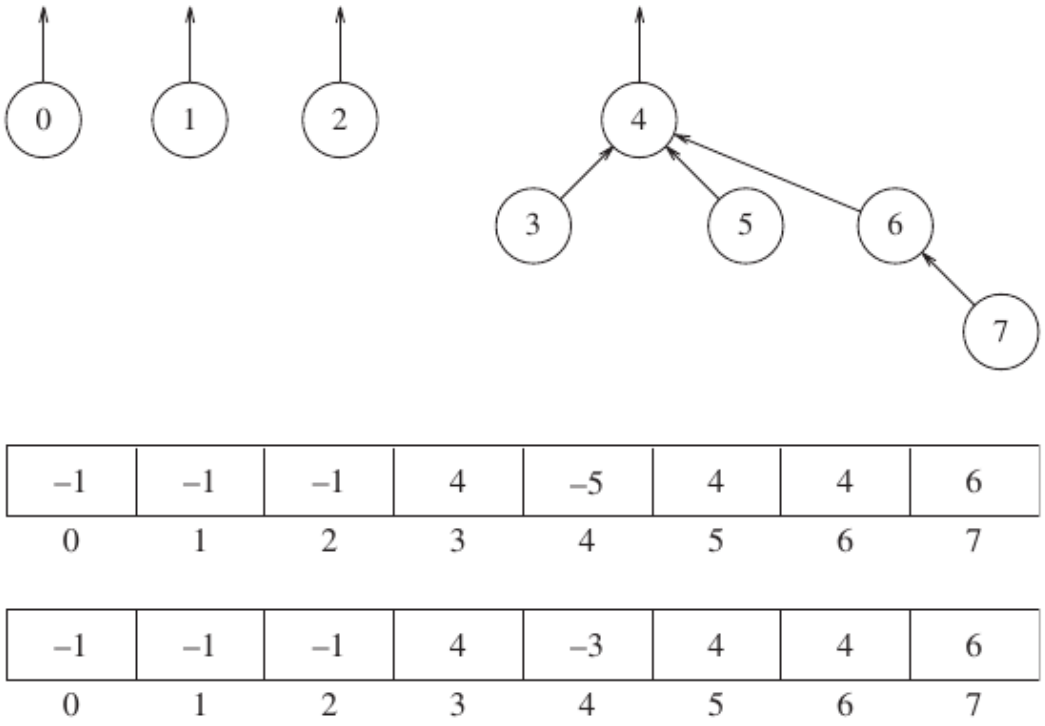


Figure 83: Forest with implicit representation for union-by-size and union-by-height

The `union` routine from before could be re-implemented as follows:

```
void DisjointSets::union( int root1, int root2 ) {
    // root2 is deeper so make root2 the new root.
    if (s[root2] < s[root1]) {
        s[root1] = root2;
    }
    else {
        // update height if both sets are the same
        if (s[root1] == s[root2]) {
            s[root1]--;
        }

        // make root1 the new root
        s[root2] = root1;
    }
}
```

16.8 Path compression

The **path compression** operation improves on the $O(M \cdot log\ N)$ worst-case `find` cost for M `find` s. The operation is performed during a `find` operation and is independent of the strategy used to perform `union`. The end goal, for `find(x)`, is to make all nodes on the path from `x` to the root to be children of the root.

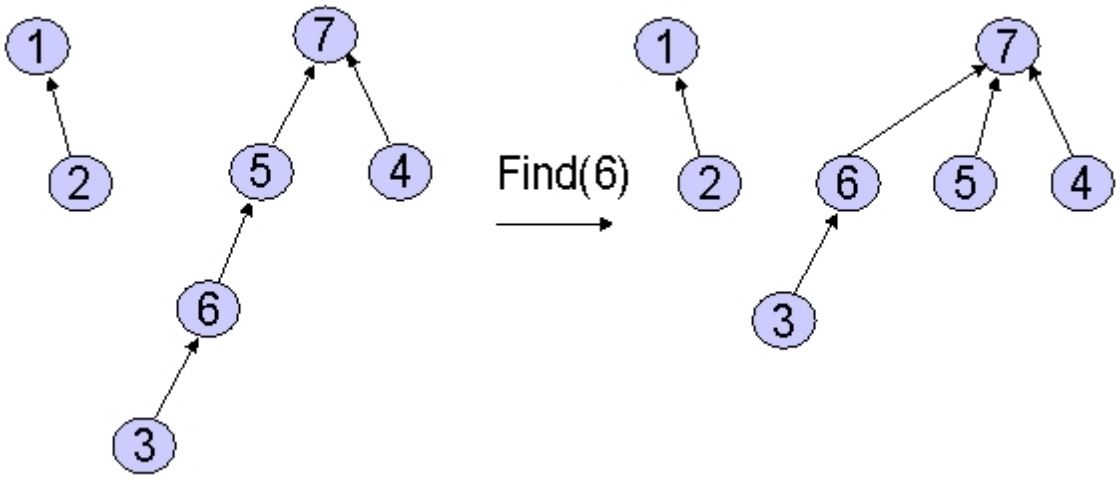


Figure 84: Path compression example

The updated code for `find` with path compression is as follows:

```
int DisjointSets::find( int x ) {
    if (s[x] < 0) { return x; }
    else          { s[x] = find(s[x]); }
}
```

Path compression is perfectly compatible with union-by-size, and thus both routines can be implemented at the same time. As for union-by-height, path compression is not entirely compatibe since it can change the heights of the tree and it's not clear how to recompute them efficiently. Then the heights stored for each tree become estimated heights (sometimes known as **ranks**), but it turns out that **union-by-rank** (which is what this has now become) is just as efficient in theory as union-by-size.

16.8.1 Union by rank

After path compression, the height of the tree may not be accurate so a height of the tree are estimated. This estimated height is known as **rank** and is a upper bound on the actual heights.

16.9 Summary

- Simple data structure to maintain disjoint sets.
- Important for graph theoretical problems.
- Union step is flexible and so we get a much more effective algorithm.
- Path compressions – earliest forms of self-adjustment
  - Seen in splay trees and skew heaps
  - Use extremely interesting because from theoretical pov one of the first algorithms that was simple with a not-so-simple worst case analysis.
- Any sequence of $M = \Omega(N)$ union-find operations takes a total of $O(MlogN)$ running time.
- Model: Union/Finds in any order Union-by-rank with path compression 41

17 Lecture 17: Graph algorithms

17.1 Graphs

A **graph** $G = (V, E)$

- consists of a set of **vertices**, V, and a set of **edges**, E.
- Each edge is a pair (v, w), where $v, w \in V$. Edges are also known as **arcs**.
- A vertex w is **adjacent** to v if and only if $(v, w) \in E$.

- Sometimes an **edge** has a third component known as either a **weight** or a **costs**.

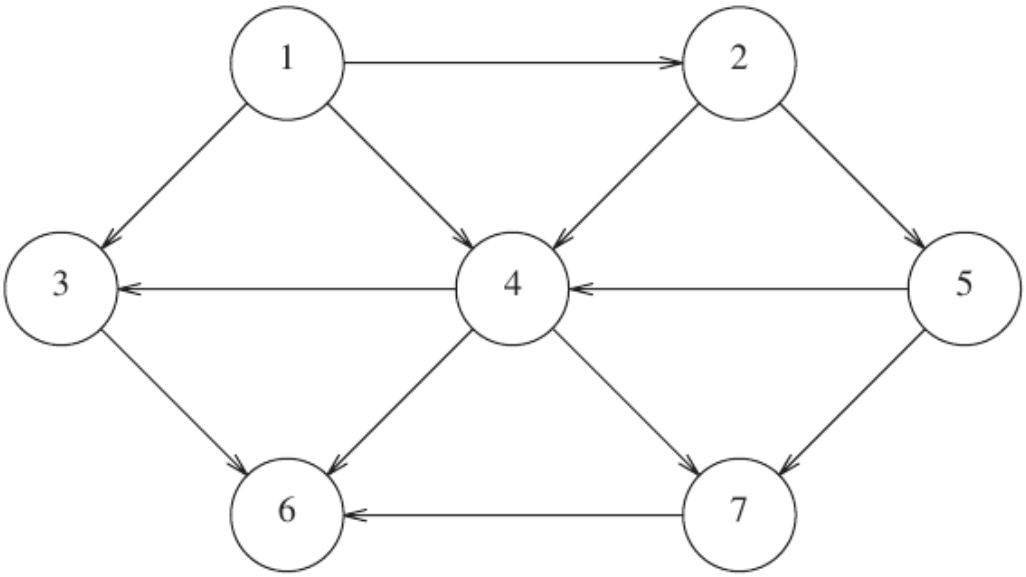


Figure 85: Example of a directed graph

17.1.1 Weighted graph

A **weighted graph** or a **network** is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand. Such graphs arise in many contexts, for example in shortest path problems such as the traveling salesman problem.

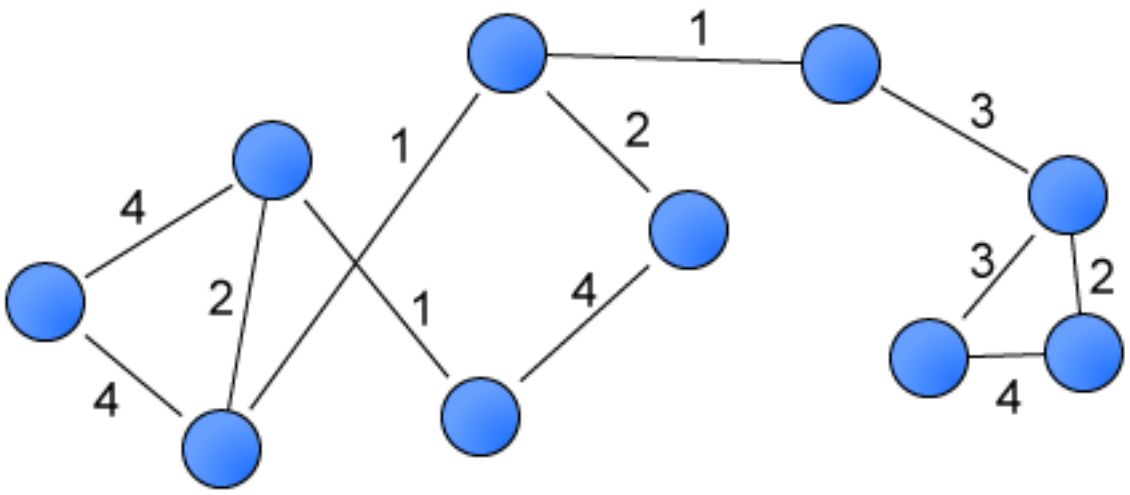


Figure 86: A weighted graph with ten vertices and twelve edges.

17.1.2 Directed vs undirected graphs

When a graph has an ordered pair of vertexes, it is called a **directed graph**. The edges of the graph represent a specific direction from one vertex to another.

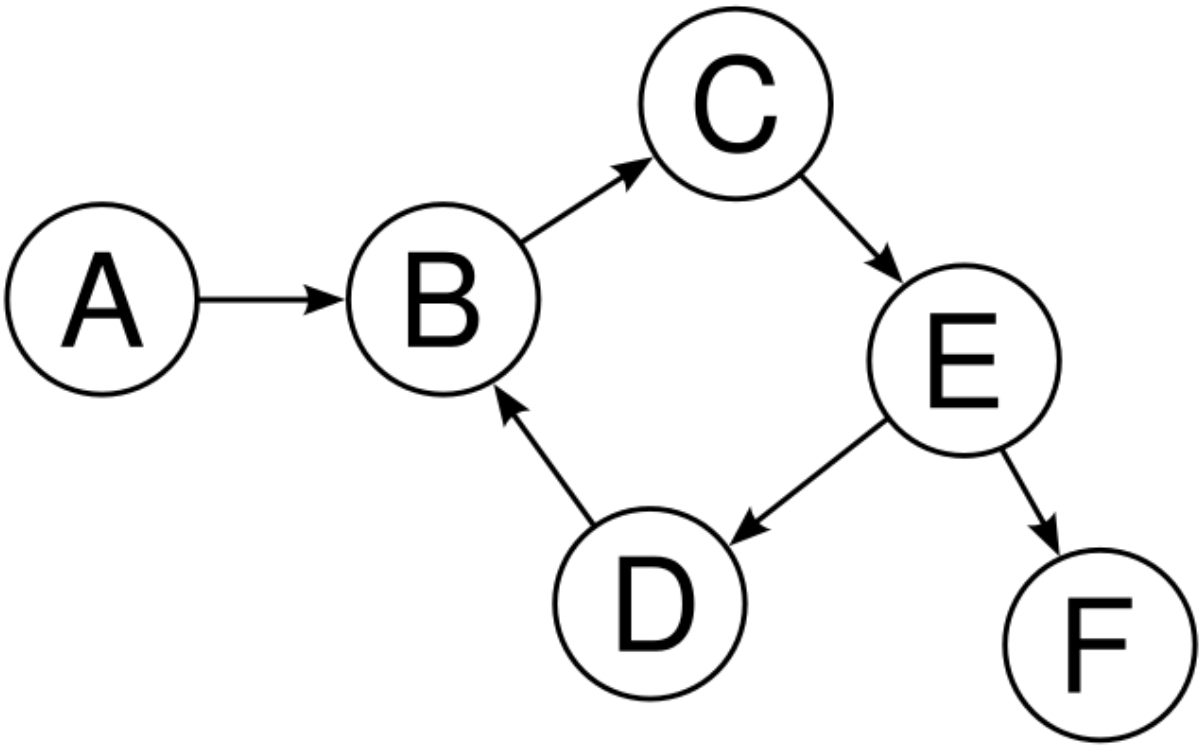


Figure 87: Example of directed graph

A real world example of a directed graph could be the relantionship between a money lender and a borrower; if any edge from a person A to a person B corresponds to A owes money to B, then B doesn't need to owe money to A because owing money is not necessarily reciprocated.

When a graph has an unordered pair of vertexes, it is an undirected graph. In other words, there is no specific direction to represent the edges. The vertexes connect together by undirected arcs, which are edges without arrows. If there is an edge between vertex A and vertex B, it is possible to traverse from B to A, or A to B as there is no specific direction.

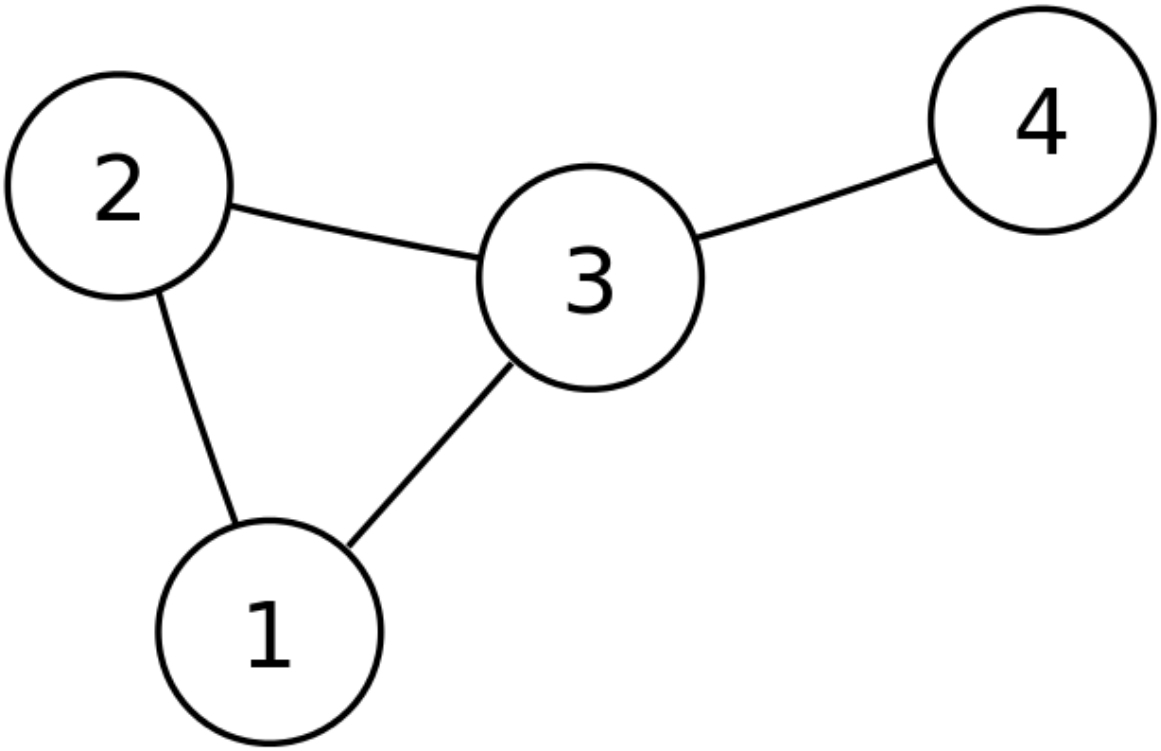


Figure 88: Example of undirected graph

A real example of a undirected graph could be a group of people at a party, where there is an edge between two people if they shake hands. Since person A can shake hands with a person B only if B also shakes hands with A, then there's no need for directions.

17.1.3 Path, length, loop,

A **path** in a graph is a sequence of vertices $w_1, w_2, \ldots, w_N$ such that $(w_i, wi + 1) \in E$ for $1 \leq i < N$.

The **length of a path** is the number of edges on the path, which is equal to $N - 1$. We allow a path from a vertex to itself, if this path contains no edges, then the path length is 0. If the graph contains an edge (v, v) from a vertex to itself, then the path v, v is sometimes referred as a **loop**.

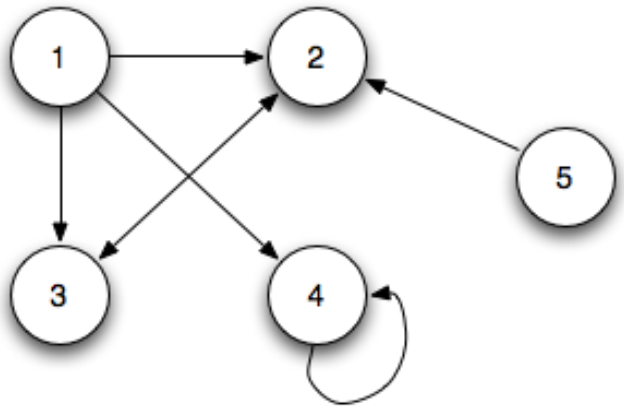
A **simple path** is a path such that all vertices are distinct, except that the first and last could be the same.

In a directed graph, a **cycle** is a path of length at least 1 such that $w_1 = w_N$; this cycle is simple is the path is simple. For undirected graphs, edges must be distinct. A directed graph is **acyclic** is it has no cycles; this graph is sometimes referred to by its abbreviation, **DAG**.

17.2 Representation of graphs

17.2.1 Adjacency matrix

- Two-dimensional array. This is known as an **adjacency matrix** representation.
- For each edge (u, v), we set $A[u][v]$ to `true`; otherwise the entry in the array is `false`.
- If the edge has a weight associated with it, then we can set $A[u][v]$ equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.
- This representation's merit lies on its extreme simplicity. However, its space requirement is $\Theta(|V|^2)$, which can be quite prohibitive if the graph doesn't have many edges.
- An adjacency matrix is an appropriate representation if the graph is **dense**, meaning $|E| = \Theta(|V|^2)$.



	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

Figure 89: An adjacency matrix representation of a graph

17.2.2 Adjacency list

- For each vertex, we keep a list of all adjacent vertices.
- The space requirement is then $O(|E| + |V|)$, which is linear in the size of the graph.
- This type of representation is good for **sparse** graphs.
- If the edges have weights, then this additional information is also stored in the adjacency lists.
- Adjacency lists are the standard way to represent graphs. Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially doubles.

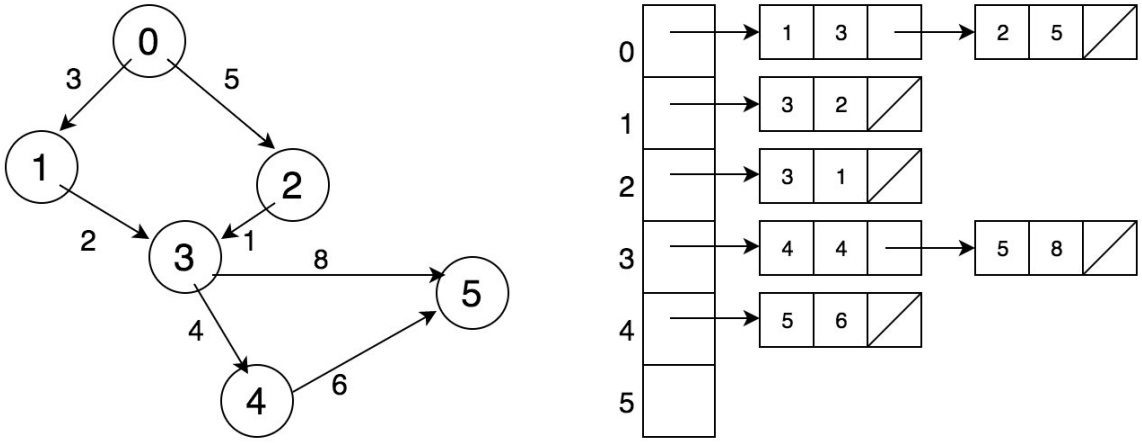


Figure 90: An adjacency list representation of a graph

17.3 Topological sort

A **topological sort** is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.

- Topological sort is defined *only* for directed graphs.
- Only possible for **DAG**; if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.
- The ordering is not necessarily unique; any legal ordering will do.

17.3.1 Finding a topological ordering

A simple algorithm to find a topological ordering is

1. Find any vertex with no incoming edges.
2. Then print this vertex, and remove it, along with its edges, from the graph.
3. Then, we apply this same strategy to the rest of the graph.

To formalize this, we define the **indegree** of a vertex v as the number of edges (u, v). We compute the indegrees of all vertices in the graph. Assuming that the indegree for each vertex is stored, and that the graph is read into an adjacency list, we can then apply the following pseudocode algorithm to generate a topological ordering:

```
/*

* findNewVertexOfIndegreeZero scans the array of vertices looking for
a vertex with indegree 0 that has not already been assigned a topological
number. It returns NOT_A_VERTEX if no such vertex exists; this indicates
    that
the graph has a cycle.

*/
void Graph::topsort():
    for (int counter = 0; counter < NUM_VERTICES; counter++):
        Vertex v = findNewVertexOfIndegreeZero()
        if (v == NOT_A_VERTEX):
            throw CycleFoundException
        v.topNum = counter

        // decrement each vertex's indegree
        for each Vertex w adjacent to v:
            w.indegree--;
```

Because `findNewVertexOfIndegreeZero` is a simple sequential scan of the array of vertices, each call to it takes $O(|V|)$ time. Since there are $|V|$ calls, the running time of the algorithm is $O(|V|^2)$. **Is it possible to do better?**

There are a few things that can improve the algorithm:

- The sequential scan through the array of vertices gives a poor running time. We can remove this inefficiency by keeping all the (unassigned) vertices of indegree 0 in a special box. Either a queue or a stack can be used to implement the box.
- Then all vertices of indegree 0 are placed on an initially empty queue.
- While the queue is not empty, a vertex v is removed, and all vertices adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0.
- The topological ordering then is the order in which the vertices dequeue.

A pseudocode implementation of the new and better algorithm could be as follows:

```
void Graph::topsort:
    Queue<Vertex> q
    int counter = 0

    q.makeEmpty()
    for each Vertex v
        if (v.indegree == 0)
            q.enqueue(v)

    until q.isEmpty():
        Vertex v = q.dequeue() # remove element
        v.topNum = counter     # assign next number

        for each Vertex w adjacent to v:
            if --w.indegree == 0:
                q.enqueue(w)

    if counter != NUM_VERTICES:
        throw CycleFoundException
```

The time to perform this algorithm is $O(|E| + |V|)$ if adjacency lists are used. This is apparent when one realizes that the body of the **for** loop is executed at most once per edge. Computing the indegrees can be done with the following code; this same logic shows that the cost of this computation is $O(|E| + |V|)$, even though there are nested loops.

The queue operations are done at most once per vertex, and the other initialization steps, including the computation of indegrees, also take time proportional to the size of the graph.

17.4 Shortest path algorithms

The input for the shortest-path problems is a **weighted graph**:

Associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge. The cost of a path $v_1 v_2 \ldots v_N$ is $\Sigma_{i=1}^{N-1} c_{i,i+1}$. This is known as the **weighted path length**. The **unweighted path length** is merely the number of edges on the path, namely, $N - 1$

> **Single-Source Shortest-Path Problem:**
>
> Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

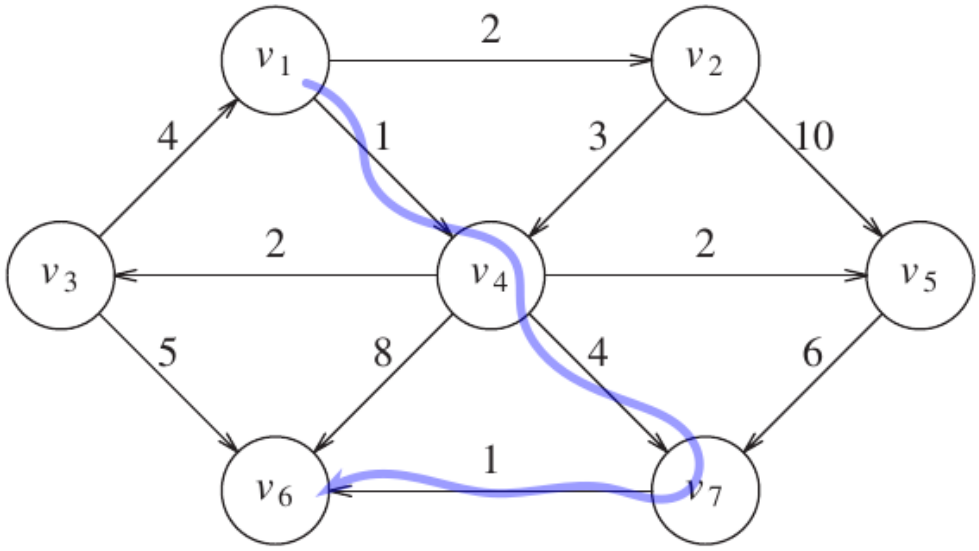


Figure 91: A directed graph G

In the directed graph G (from the figure), the shortest weighted path from v_1 to v_6 has a weight (or cost) of 6 which goes from v_1 to v_4 to v_7 to v_6. The shortest unweighted path between these two vertices is 2 (number of edges $N = 3$, $N - 1 = 3 - 1 = 2$). Notice graph G has no edges of negative weight.

Negative edges can cause a problem known as a **negative-cost cycle**:

- Always ever shortest path by following loops that are arbitrarily long. This causes the shortest path between two points to be undefined.
- When a negative-cost cycle is present in the graph, the shortest paths are not defined.
- Negative-cost edges aren't necessarily bad, as the cycles are, but their presence seems to make the problem harder.

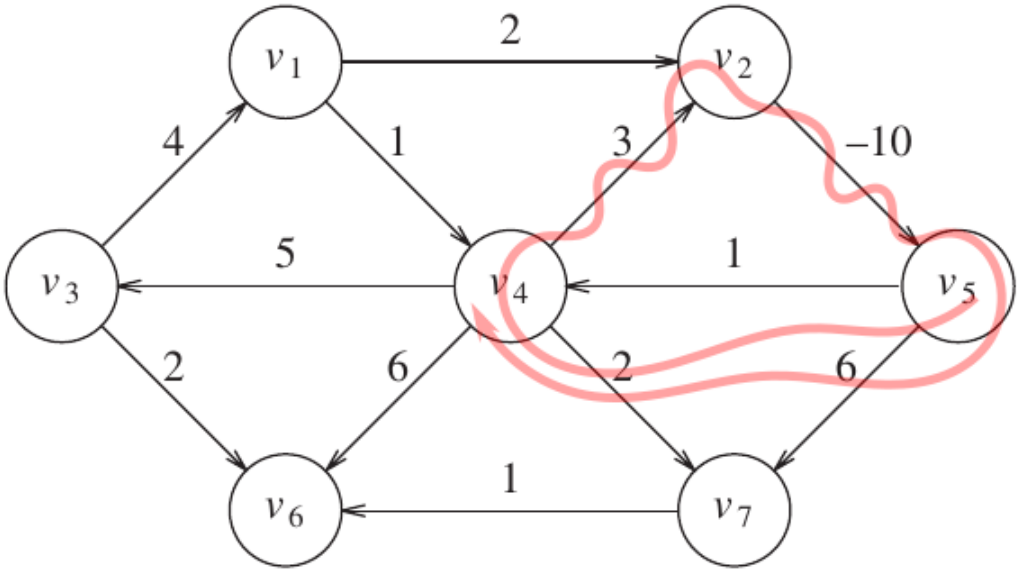


Figure 92: A graph H with a negative-cost cycle

In the graph H, the path v_5 to v_4 has cost 1, but a shorter path exists by following the loop v_5, v_4, v_2, v_5, v_4, which has cost -5. However, the path isn't still the shortest path because we could still in the loop arbitrarily long. Thus, the shortest path between these two points is undefined.

We can model airplane or other mass transit routes by graphs and use a shortest- path algorithm to compute the best route between two points. In this and many practical applications, we might want to find the shortest path from one vertex, s, to only one other vertex, t. Currently there are no algorithms in which finding the path from s to one vertex is any faster (by more than a constant factor) than finding the path from s to all vertices.

17.4.1 Unweighted shortest paths

For this, we consider an unweighted graph K which is an input parameter, and we would like to find the shortest path from s to all vertices. We are only interested in the number of edges contained on the path, so *there are no weights on the edges*.

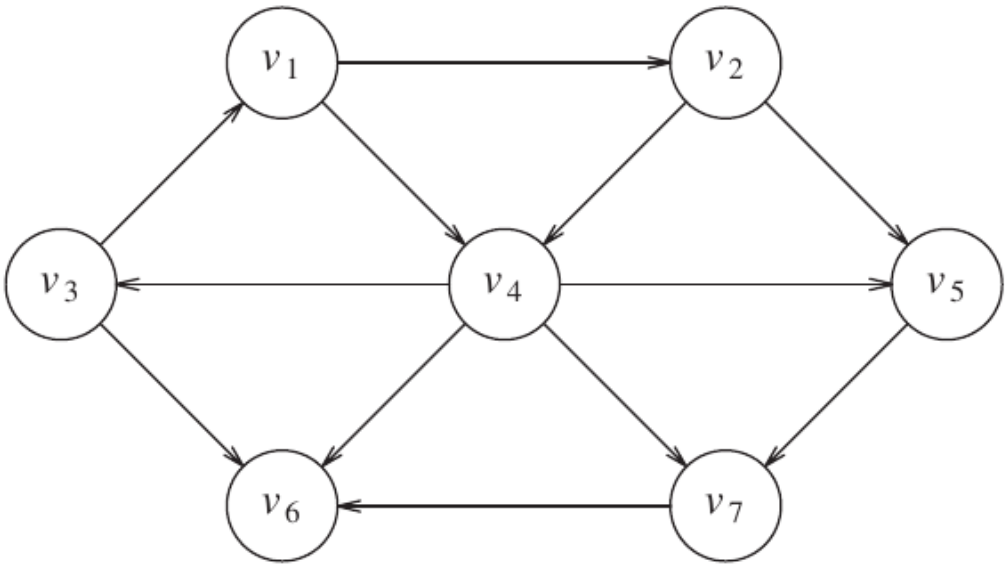


Figure 93: An unweighted directed graph K

Suppose we choose s to be v_3:

- Immediately, we can tell that the shortest path from s to v_3 is then a path of length 0. We can mark this information on the vertex v_3 by placing a 0 beside it.
- Now we can start looking for all vertices that are a distance 1 away from s. These can be found by looking at the vertices that are adjacent to s. If we do this, we see that v_1 and v_6 are one edge from s. Thus, v_1 and v_6 has a 1 beside them.
- We can now find vertices whose shortest path from s is exactly 2, by finding all the vertices adjacent to v_1 and v_6 (the vertices at distance 1), whose shortest paths are not already known. This search tells us that the shortest path to v_2 and v_4 is 2.
- Finally we can find, by examining vertices adjacent to the recently evaluated v_2 and v_4, that v_5 and v_7 have a shortest path of three edges. All vertices have now been calculated, and we end up with graph K', which is final result of the algorithm.

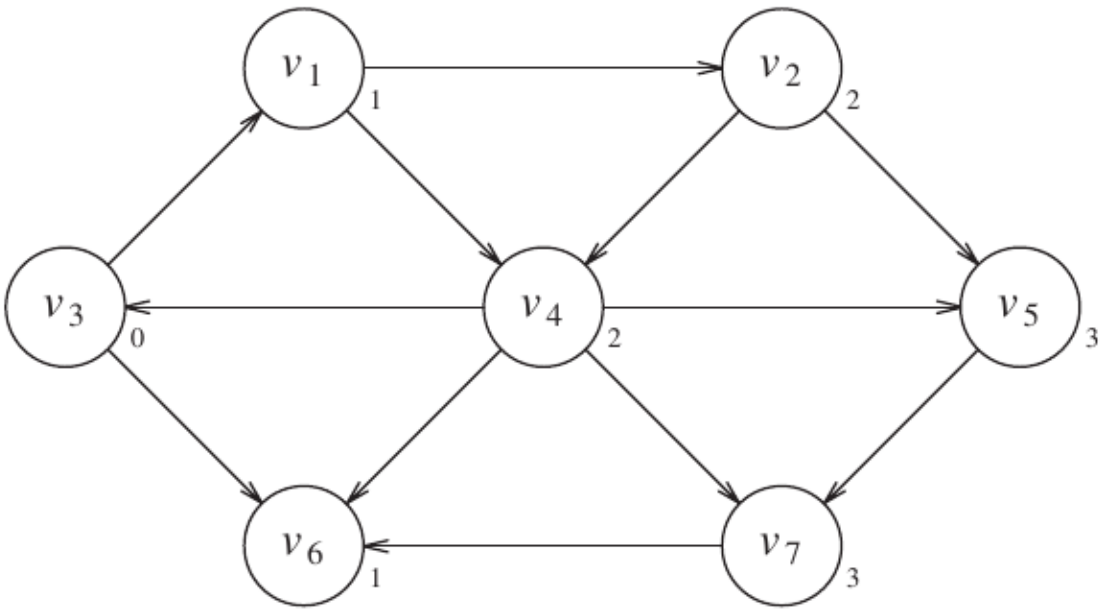


Figure 94: Graph K' with final shortest paths

This strategy for searching a graph is known as **breadth-first search**. It operates by processing vertices in layers: The vertices closest to the start are evaluated first, and the most distant vertices are evaluated last. This is much the same as a level-order traversal for trees. Translated to pseudocode, the basic algorithm looks as follows:

```
void Graph::unweighted( Vertex s ):
    # For each vertex, we will keep track of three pieces of information.
    # First, we will keep its distance from s in the entry v. Initially
    # all vertices are unreachable except for s, whose path length is 0.
    for each Vertex v:
        v.dist = INFINITY
        v.know = false

    # set s's path length
    s.dist = 0

    for (int currDist = 0; currDist < NUM_VERTICES; currDist++):
        for each Vertex v:
            if !v.known && v.dist == currDist:
                # vertex processed so mark as done
                v.known = true;
                for each Vertex w adjacenct to v:
                    if w.dist == INFINITY:
                        w.dist = currDist + 1
                        w.path = v
```

The running time of the algorithm is $O(|V|^2)$, because of the doubly nested for loops. However, there are still some inefficiencies and the algorithm could be improved further.

17.4.2 Dijkstra's Algorithm

If the graph is weighted, the problem (apparently) becomes harder, but we can still use the ideas from the unweighted case.

We keep all of the same information as before. Thus, each vertex is marked as either *known* or *unknown*. A tentative distance d_v is kept for each vertex, as before. This distance turns out to be the shortest path length from s to v using only known vertices as intermediates. As before, we record p_v, which is the last vertex to cause a change to d_v.

The general method to solve the single-source shortest-path problem is known as **Dijkstra's algorithm**, which is a prime example of a greedy algorithm. Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.

The pseudocode for the Dijkstra's algorithm is as follows:

```
void Graph::dijkstra( Vertex s ):
    for each Vertex v:
        v.dist = INFINITY
        v.known = false

    s.dist = 0

    while there is an unknown distance vertex:
        Vertex v = smallest unknown distance vertex
        v.known = true

        for each Vertex w adjacent to v:
            unless w.known:
                DistType cvw = cost of edge from v to w
                if v.dist + cvw < w.dist:
                    # update w
                    decrease(w.dist to v.dist + cvw)
                    w.path = v
```

18 Lecture 18: Graph algorithms (cont.)

18.1 Weighted shortest path: Dijkstra

If the graph is weighted, the problem (apparently) becomes harder, but we can still use the ideas from the unweighted case.

We keep all of the same information as before. Thus, each vertex is marked as either *known* or *unknown*. A tentative distance d_v is kept for each vertex, as before. This distance turns out to be the shortest path length from s to v using only known vertices as intermediates. As before, we record p_v, which is the last vertex to cause a change to d_v.

The general method to solve the single-source shortest-path problem is known as **Dijkstra's algorithm**, which is a prime example of a greedy algorithm. Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage.

18.1.1 General description

Suppose we want to find a shortest path from a given nodesto other nodesin a network (one-to-all shortest path problem).

- Dijkstra's algorithm solves such a problem:
  - It finds the shortest path from a given node s to all other nodes inthe network.
  - Node s is called a starting node or an initial node.
- How is the algorithm achieving this?
  - Dijkstra's algorithm starts by assigning some initial values for the distances from node s and to every other node in the network.
  - It operates in steps, where at each step the algorithm improves the distance values.
  - At each step, the shortest distance from node s to another node is determined.

18.1.2 Formal description

The algorithm characterizes each node by its state; the state of a node consists of two features:

- **distance value**, a scalar representing an estimate of its distance from node s.
- **status label**, an attribute specifying whether the distance value of a node is equal to the shortest distance to node s or not.
  - The status label of a node is **permanent** (or known) if its distance is equal to the shortest distance from node s.
  - Otherwise, the status label of a node is **temporary** (or unknown).

The algorithm maintains and step-by-step updates the states of the nodes. At each step, one node is designate as **current**.

18.1.3 Notation

We'll use the following notation for the rest of the text:

- d_ℓ denotes the distance value of a node ℓ.
- p or t denotes the status label of a node, where p stands for permanent and t stands for temporary.
- c_{ij} is the cost of traversing link (i, j) as given by the problem.

The state of a node ℓ is the ordered pair of its distance value d_ℓ and its status label.

18.1.4 Algorithm steps

Step 1. *Initialization*

- Set node s to distance value 0, and label it as **permanent**. Thus, the state of node s is $(0, p)$.
- Assign to every node a distance value of ∞ and label them as **temporary**. Thus, the state of every other node (∞, t).
- Designate the node s as the **current** node.

Step 2. *Distance Value Update and Current Node Designation Update*

Let i be the index of the current node.

1. Find the set J of **nodes with temporary labels** that can be reached from the current node i by a link (i, j). **Update the distance values of these nodes**.
    - For each $j \in J$, the distance value d_j of node j is updated as follows new $d_j =$ min $\{d_j, d_i + c_{ij}\}$, where c_{ij} is the cost of link (i, j), as given in the network problem.
2. Determine a node j that has the smallest distance value d_j among all nodes $j \in J$, find $j*$ such that $\min_{j \in J} d_j = d_{j*}$.
3. Change the label of node $j*$ to **permanent** and designate this node as the **current node**.

Step 3. *Termination Criterion*

- If all nodes that can be reached from node s have been permanently labeled, then stop - we are done.
- If we cannot reach any temporary labeled node from the current node, then all the temporary labels become permanent - we are done.
- Otherwise, go to Step 2.

18.1.5 Example

Given the graph M, we want to find the shortest path from node 1 to all other nodes using Dijkstra's algorithm.

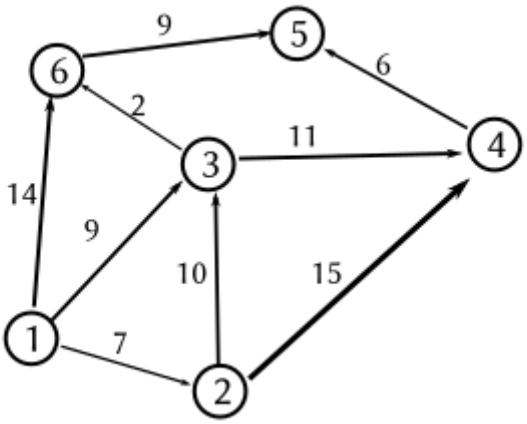


Figure 95: Graph M

Step 1:

- Node 1 is designated as the current node.
- The state of node 1 is $(0, p)$.
- Every other node has state (∞, t).

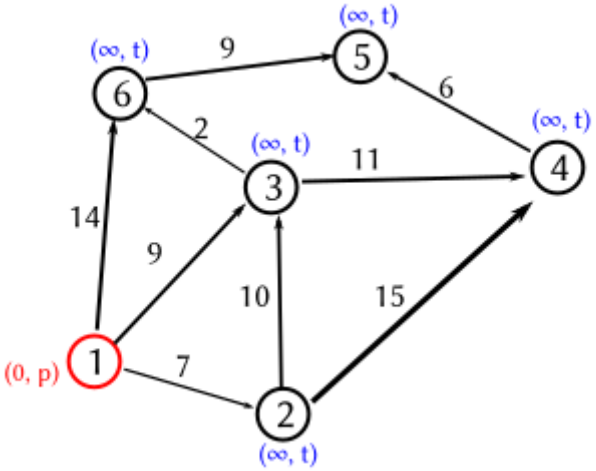


Figure 96: Graph M after initialization

Step 2:

- Nodes 2, 3, and 6 can be reached from the current node 1.
- Update distance values for these nodes:
  - $d_2 = min\{\infty, 0 + 7\} = 7$
  - $d_3 = min\{\infty, 0 + 9\} = 9$
  - $d_6 = min\{\infty, 0 + 14\} = 14$
- Now, among the nodes 2, 3, and 6, node 2 has the smallest distance value.
- The status label of node 2 changes to permanent, so its state is $(7, p)$, while the status of 3 and 6 remains temporary.
- Node 2 becomes the current node.

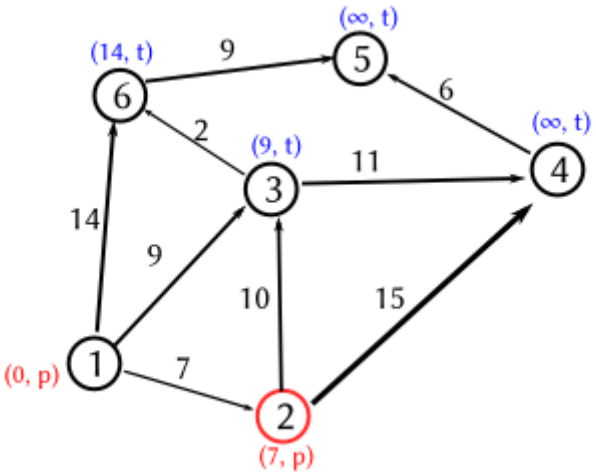


Figure 97: Graph M: Node 2 changes to permanent and becomes the current node

Step 3:

We are not done, not all nodes have been reached from node 1, so weperform another iteration (back to Step 2).

Step 2 (again):

- Nodes 3 and 4 can be reached from the current node 2.
- Update distance values for these nodes:
  - $d_3 = min\{9, 7 + 10\} = 9$
  - $d_4 = min\{\infty, 7 + 15\} = 22$
- Now, between the nodes 3 and 4, node 3 has the smallest distance value.
- The status label of node 3 changes to permanent, while the status of 6 remains temporary.
- Node 3 becomes the current node.

We're not done (Step 3 fails), so we perform another Step 2.

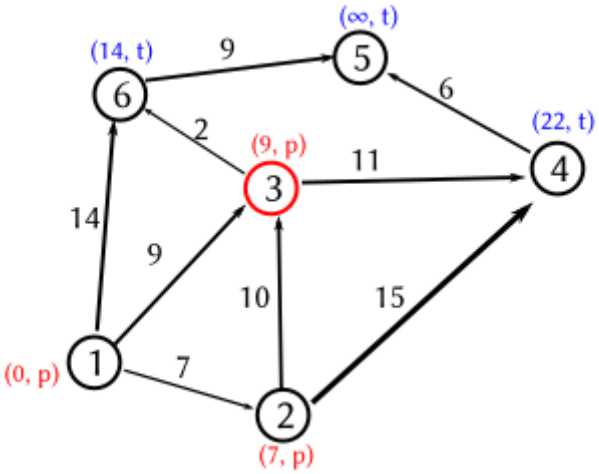


Figure 98: Graph M: Node 3 changes to permanent and becomes the current node

Step 2 (again):

- Nodes 6 and 4 can be reached from the current node 3.
- Update distance values for these nodes:
  - $d_4 = min\{22, 9 + 11\} = 20$
  - $d_6 = min\{14, 9 + 2\} = 11$
- Now, between the nodes 6 and 4, node 6 has the smallest distance value.
- The status label of node 6 changes to permanent, while the status 4 remains temporary.
- Node 6 becomes the current node.

We're not done (Step 3 fails), so we perform another Step 2.

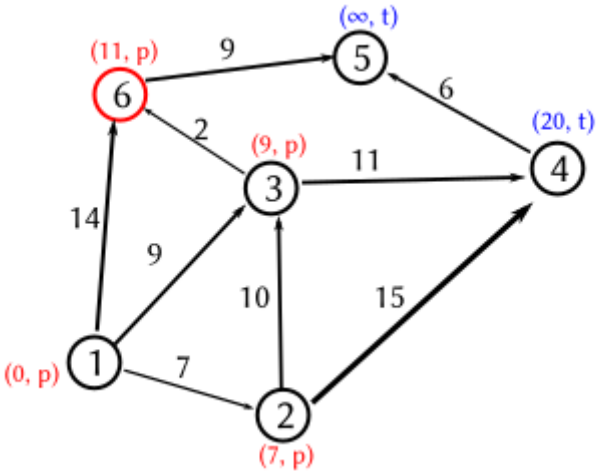


Figure 99: Graph M: Node 6 changes to permanent and becomes the current node

Step 2 (once again):

- Node 5 can be reached from the current node 6.
- Update distance value for node 5:
  - $d_5 = min\{\infty, 11 + 9\} = 20$
- Now, node 5 is the only candidate, so its status changes to permanent.
- Node 5 becomes the current node.

From node 5 we cannot reach any other node. Hence, node 4 gets permanently labeled and we're done.

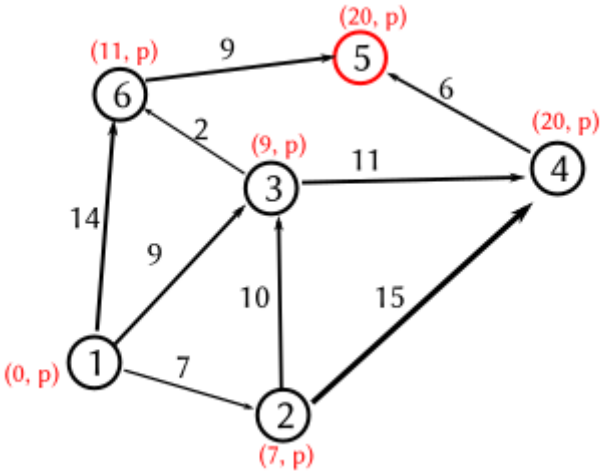


Figure 100: Graph M: Both 5 and 4 changes to permanent

18.1.6 Importance of Dijkstra's algorithm

Many more problems than you come across can be cast as shortest path problems, which make Dijkstra's algorithm a powerful and general tool. For example:

- Dijkstra's algorithm is applied to automatically find directions betweenphysical locations, such as driving directions on websites like Mapquestor Google Maps.
- In a networking or telecommunication applications, Dijkstra's algorithm has been used for solving the min-delay path problem (which is the shortest path problem). For example in data network routing, the goal is to find the path for data packets to go through a switching network with minimal delay.
- It is also used for solving a variety of shortest path problems arising in plant and facility layout, robotics, transportation, and VLSI (Very Large Scale Integration) design.

18.2 Graphs with negative edge costs

18.3 Acyclic graphs

18.4 Network flow problems