Data Structures and Algorithms

Lecturer: Dr. Junbin Gao

Contents

1 Abstract Data Types 1

1.1 Review on C++ Programming ... 1
1.2 Basic Concepts ... 3

2 C++ Classes 8

2.1 Classes ... 8
2.2 Constructors and Destructors ... 10
2.3 Functions and Class Member Functions ... 11
2.4 Constant and Static Members ... 14
2.5 Operator Overloading ... 16

3 The Bag Class 21

3.1 The Bags ... 21
3.2 Class Design ... 23
3.3 General Rules ... 26

4 The Sequence Class 28

4.1 The Sequences ... 28
4.2 The Design of Sequence Classes ... 28
4.3 The Implementation of Sequence Classes ... 30

5 Pointers and Dynamic Arrays 33

5.1 Pointers ... 33
5.2 Arrays and Array Pointers ... 34
5.3 Function Pointers ... 36

6 Dynamic Implementation of Container Classes 39

6.1 The Bag Class with A Dynamic Array . . . 39
6.2 General Rules for Class with Pointer Member . . . 43

7 Linked Lists and Their Applications 45

7.1 Lists . . . 45
7.2 General Linked Lists . . . 46
7.3 Application: Large Integers & Lists . . . 49

8 More About Linked Lists 52

8.1 Other Techniques for Lists . . . 52
8.2 One More Example: Binary Encoding . . . 53

9 Templates 55

9.1 Template Functions . . . 55
9.2 Template Classes . . . 56
9.3 Applying Template Classes . . . 59

10 Templates and Iterators 62

10.1 Node Template Classes . . . 62
10.2 STL and Their Iterators . . . 63
10.3 Node Iterator — Linked List . . . 65

11 Complexity Analysis of Algorithms 67

11.1 big-oh . . . 67
11.2 Main Properties of big-oh . . . 69
11.3 Examples . . . 70

12 Recursion and Iteration 73

12.1 Recursive Functions . . . 73
12.2 Examples . . . 75
12.3 Recursion vs. Iteration . . . 78
12.4 Backtracking Algorithms . . . 78

13 Algorithms for Search 81

13.1 Why Search? 81
13.2 Sequential Search 82
13.3 Binary Search 84
13.4 Alternative Binary Search 86
13.5 Pros and Cons of Searching Techniques 87

14 Hashing and Complexity 89

14.1 Hashing 89
14.2 Hash Tables 90
14.3 Collision Problems 92
14.4 Bucket Technique and Chained Addressing 95

15 Stacks and Their Application 98

15.1 Stack Structure 98
15.2 Applications 100

16 Queues and Their Application 102

16.1 Queues 102
16.2 Queue Applications 104

17 Dynamic Queues 107

17.1 Linked List Implementation of a Queue 107
17.2 Priority Queues 110

18 Trees and Binary Trees 111

18.1 The Tree Structures 111
18.2 Tree Representations 113
18.3 Traversing Binary Trees 115

19 Binary Search Trees and B-Trees 118

19.1 Binary Search Trees 118
19.2 Main Operations on Binary Search Trees 119
19.3 B-Trees 122
19.4 The Set Class for B-tree Nodes 123

19.5 B-tree Operations 124

20 Quadratic Sorting 127

20.1 Concepts Related to Sorting 127
20.2 Insertion Sort 128
20.3 Selection Sort 131
20.4 Comparing Insertion and Selection Sort 133

21 Recursive Sorting 135

21.1 Mergesort 135
21.2 Algorithm 136
21.3 Quicksort 139

22 Heap Sort 144

22.1 Complete Binary Trees 144
22.2 Heaps 146
22.3 Algorithms 147
22.4 Characteristics of Heapsort 151

23 Graphs 152

23.1 Graph Definition 152
23.2 Graph Implementation 153
23.3 Member Functions for a Graph Class 155

24 Algorithms for Graphs 157

24.1 Graph Traversals 157
24.2 Path Algorithms 159

Chapter 1

Abstract Data Types

This chapter marks an important step in your exploration of computer science. Up to this point, it has been difficult to divorce your study of computer science from the learning of C++. You have developed problem-solving skills, but the problems we have encountered have been very focused. That is, the problems were chosen specifically to illustrate a particular feature of C++. This is the way problem-solving skills must be developed: Start with small problems and work toward large ones.

By now, you are familiar with many features of the C++ programming language. You are ready to direct your attention toward larger, more complex problems that require you to integrate many of the particular skills you have developed. Now our attention will be directed more toward issues of algorithm design and data structure and less toward describing C++. If we need a particular feature of C++ that you have not yet learned, we will introduce it when appropriate.

1.1 Review on C++ Programming

1.1.1 Good Programming Practices

- *Defensive Programming (Robustness):* A program ...
  - should not crash on *incorrect* input/should always check input for correctness.
  - should always check the returned value of `new` operator (trying to access a null pointer will cause your program to crash)
  - should always crash gracefully (i.e. with error message, not segmentation fault).
- *User Interface:* A program ...
  - should use `cin` and `cout` for user interface.
  - always prompt the user for input. For instance,

```
cout << "Enter an integer value: ";
cin >> i;
```

- always annote your output. For example,

```
cout << "The value of i is: " << i << endl;
```

- *Programming Style:* A program should ...
  - be one statement per line.
  - use descriptive variable and function names.
  - use uppercase and lowercase letters to improve readability.

```
int arraySum(int* array, int length);
```

  - use #define statements for constant definitions.

```
#define MAX_LENGTH 20
#define MONTHS_IN_YEAR 12
```

  - use a *clear* and *uniform* indentation scheme.

```
#include <iostream.h>

int main() {
  int num;
  cout << "Enter an integer: ";
  cin >> num;
  if (num == 10)
    cout << "The number is 10" << endl;
  else
    cout << "The number is not 10" << endl;
}
```

- *Program Documentation:* always write comments or documents for programs.

1.1.2 Object Oriented Paradigm

- *Object-oriented programming (OOP)* is a programming paradigm that takes the abstract data model to a higher level of abstraction. In OOP, an *object* is an entity consisting of a data structure and its associated operations or *methods*.
- The three main characteristics of OOP are encapsulation, inheritance and polymorphism.
  - *Encapsulation* is combining the data and methods that manipulate the data in one place — an *object*. All processing for an object is bundled together in a library and hidden from the user.
  - *Inheritance* is the ability to define an object as a descendant of an existing object.

- *Polymorphism* occurs when a single method name, used up and down the hierarchy of objects, acts in an appropriate—possibly different—way for each object.

1.2 Basic Concepts

1.2.1 Abstract Data Types (ADT)

- A *data type* consists of two parts, a set of data items and fundamental operations on this set. We can see that the *integer type* consists of values (whole numbers in some defined range) and operations (addition, subtraction, multiplication and division etc).
- When using an **ADT** we need only worry about *what* each operation does, not *how*.
- A *Data Structure* is an aggregation of simple and composite data type into a set with defined relationship. The *structure* means a set of rules that hold the data together. In other words, if we take a combination of data types and fit them into a structure such that we can define its relating rules, we have made a data structure.
- The examples: Array and Record.
- An *Algorithm* is a precise, step-by-step method of doing a task in a *finite amount of time*.

1.2.2 Data Types in C++

- Primary types:

```
long int l = 0xABCDEF01L;
char     c;
double   d;
```

- Pointers:

```
char   *cp = "Hello";
short  *sp;
```

- Enumerated types:

```
enum weeks {SUN, MON, TUE, WED, THU, FRI, SAT};
week Today = MON;
```

- Boolean Data Type: C++ provided a new built-in data type called `bool` (boolean type). A boolean type variable takes one of only two constant values `true` and `false`.

```
bool GuessIt;
GuessIt = true;
```

- Structures:

```
struct person {
  int age;
  char *firstName;
  char *lastName;
};
person bob = {33, "Bob", "Smith"};
```

- Unions:

```
union charOrInt {
  char c;
  int i;
};

charOrInt coi = {'a'}, coi2 = {10};
coi2.c = coi.c;
```

- Arrays:

```
person people[10];
int a[] = {1, 2, 3, 4, 5};
```

1.2.3 Pointers in C++

- A *pointer* does not *hold* any data but instead *holds* the address of memory location of data. Care should be taken to make sure that a pointer does actually point to data of the correct type. Failure to do so may result in a segmentation fault.
- Incorrect use of pointers can result in your program crashing.
- Pointer examples:

```
int     *iptr;
char    *cptr;
```

1.2.4 Constant pointers

- In C++ a constant variable is a variable whose value can't be changed. For pointers the idea of constants is more complex because we must specify whether it is the pointer itself that is constant, or the data it points to.
- Four different types of constant pointers:
  - non-constant pointer to non-constant data

    ```
    int* p = &i;
    ```

  - non-constant pointer to constant data

    ```
    const int* p = &i;
    ```

  - constant pointer to non-constant data

    ```
    int* const p = &i;
    ```

  - constant pointer to constant data

    ```
    const int* const p = &i;
    ```

- This can be extended to pointers to pointers, etc.

  ```
  const int* const* const pp = &p;
  ```

  You can read the above definition as

  ```
  const int* (const* (const pp)) = &p;
  ```

  which means that pp is a constant pointer pointing to a pointer whose *type* is constant pointer to a constant *int* variable. Complicated?? Yes ...

1.2.5 Example: Swap using pointers

- Incorrect version:

  ```
  void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
  }
  ```

Why? The `swap` function exchanges the values of local copies of its arguments `a` and `b` other than themselves. Please read and run this program at http://turing.une.edu.au/~comp282/Lectures/Lecture_01/Examples/test_swap.cpp

The default initialization method of argument passing in C++ is to copy the values of arguments into the storage of the parameters. This referred to as *pass-by-value*.

Under pass-by-value, the function never accesses the arguments of the call. The values that function manipulates are its own local copies; they are stored on the run-time stack. When function terminates, these values are lost.

- Correct version with pointers:

```
template <class T>
void swap(T *x, T *y) {
  T temp = *x;
  *x = *y;
  *y = temp;
}
```

Now the values have been exchanged. See the program http://turing.une.edu.au/~comp282/Lectures/Lecture_ 01/Examples/test_refswap.cpp

1.2.6 Pass By Reference

- C++ provides us with another way of passing addresses to functions via the use of reference variables. These variables act like pointers but don't look like them.
- Examples:

```
int i;
int &iref = i;  // Reference variable
```

- *Pass-by-reference* is the technique of making a variable accessible to a function by passing its address.
- A correct `swap` function using call-by-reference:

```
void swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

- When the parameters are references, the function receives the *lvalue* of the argument rather than a copy of its value. This means that the function knows where the argument resides in memory and can therefore change its value or take its address.

- Any use of the pass-by-reference parameter within the body of the function will access the argument in the calling program
- Pass-by-reference is one of very powerful ways in passing values to a function. It has the similar definition format as the pass-by-value, but its functioning is more likely to that of pointer version.
- If the function does not change the value of the argument within the body of the function, for the sake of security you can declare the parameter type as *const reference parameter*. For example

```
int difference(const int & p1, const int &p2);
```

1.2.7 `new` & `delete` Operators

- We use the `new` and `delete` operators to allocate and free memory in C++. C programmers can think of these as `malloc` and `free` in C respectively.
- The `new` operator allocates memory for any type including arrays and returns a pointer to the desired type.
- In other words, the `new` operator creates a new dynamic variable of any types. The dynamic variable does not have any identifiers or names but we know the address of the variable through the returned appointer to the variable.
- The creation of new dynamic variables is called memory *allocation* and the memory is *dynamic memory*.
- The dynamic variables created by the `new` operator must be freed or destroyed by the `delete` operator explicitly. It is desired for you always to destroy the dynamic memory if you don't want it any more or before terminating the program.
- The `delete` operator frees memory for all types except arrays.
- For arrays, we must use the `delete[]` operator.
- Examples:

```
int *iptr = new int;
int *iiptr = new int(1024);
// Put an initial value in the dynamic variable
char *cptr = new char[1024];

delete iptr;
delete iiptr;
delete [] cptr;
```

Chapter 2

C++ Classes

2.1 Classes

2.1.1 Concept of C++ class

- C++ classes are similar to structures in C, with the main difference being that classes can have functions, or *methods*, as well as variables, or *data members* in their definitions.
- The other main difference is that classes use the technique of *information hiding* to avoid incorrect use of the class. This is done via the `public`, `private` and `protected` key words.
- A *class* can be considered a type, whereas an *object* is an instance of a class. For example, the primary type `int` of C++ is a class, any variables declaration as `int` type can be considered as `int` objects. In the following small section of pseudo-code, `Date` is a class, and `dObj` is an object.

```
class Date {
    ...
    //Definition
    ...
};

Date dObj;
```

- In C++ the `class` and `struct` keywords are interchangeable.
- The `class` technique of C++ is a good tool used for implementing the concept of abstract data types in data structures

2.1.2 A Class Outline

- The general syntax of declaring a class

```
class class_name {
   public:
      Function prototypes that can be used
      outside the class
      ......
   private:
      Type definitions and member declarations
      accessible only inside the class
      ......
};
```

- A class declaration consists of three parts:
  1. The class *head*: includes the C++ keyword `class` and the *name* of the new class
  2. The public section: begins with the C++ keyword `public` followed by a colon. A list of items appears after the colon. These are items that are made available to anyone who uses the new data type.
  3. The private section: begins with the C++ keyword `private` followed by a colon. After a colon is a list of items that are part of the class but are not directly available to programmers who use the class.

2.1.3 Example Class and Its Usage

- Let us define a class

```
class Card {
public:
  Card(face_t f = ace,
       suit_t s = hearts); // constructors
  Card(const Card &c);

  Card &operator=
       (const Card &c);    // operator

  ~Card();                 // destructor

  face_t getFace() const;  // methods
  suit_t getSuit() const;

private:
  face_t face;             // data members
  suit_t suit;
};
```

- Then we use this class just like using the standard type `int`

```
Card c;
Card *cp = new Card(c);
```

2.2 Constructors and Destructors

2.2.1 Constructors

- Constructors are used to initialize an object. They are called automatically whenever a new object is created. If no constructor is given, the compiler will automatically generate a *default* one.
- Constructors always have the same name as the class, and never have a return type (not even void: you must not write `void` at the front of the constructor's head).
- Constructors are declared like other methods with the above differences.
- You may declare as many constructors as you like — one for each different way of initializing an object.
- Two special constructors:
  1. Default constructor: If you write a class with no constructors, then the compiler automatically creates a simple default constructor. This automatic default constructor doesn't do much work. It is a good way to write your own constructors even your own default constructor, rather than depending on the automatic default constructor. Your own constructor is the one which is used by programmers to declare a variable of the class without having to provide any arguments for the constructor.
  2. Copy constructor: Copy constructor is a special member function which will be called when the program creates a new object through object assignment or assigns an object to another. We will talk about it later.

2.2.2 Destructors

- Destructors are used to do any final preparations before an object is destroyed. They are called automatically, either when `delete` is used on a dynamically allocated object, or when a static object goes out of scope.
- Unlike constructors, there can only be one destructor per class. The name of a destructor must be the class name preceded by a tilde (˜). Destructors have no arguments, and no return type.
- Destructors are particularly useful for freeing any dynamically allocated memory the object might have. Without appropriate destructors, your program may have memory leaks (this isn't Java!).

2.2.3 Example: How constructor and destructor work?

- What is the output of this program?

```
class A {
public:
   A() { cout << "Making A"   << endl; }
  ~A() { cout << "Breaking A" << endl; }
};

class B {
public:
   B() { cout << "Making B"   << endl; }
  ~B() { cout << "Breaking B" << endl; }

private:
  A a;
};

void main() {
  B b;
}
```

- Answer:

```
Making A
Making B
Breaking B
Breaking A
```

- Why? All the data members of an object are initialized before the constructor builds up the object itself. In the above example, the program declares a new object `b` of class `B`. Before building up the object `b` itself, its only data member, the object `a` of class `A`, should be initialized or created first. Thus in order for the program to build up the object `a`, the constructor `A()` of class `A` is called first. That is why we see the output `Making A` first and then `Making B`.

  Before exiting the program, the object `b` is destroyed by calling its destructor first and the data member `a` of the object `b` by calling its destructor, respectively.

  Surprise you!? Try the program yourself.

2.3 Functions and Class Member Functions

2.3.1 Function and Its Type

- Either a function's name or any parameter's name is not part of its type.

- A function's type is determined only by its return type and its parameter list including parameter's type on the list.
- Example:

```
int lexicoCompare(const string &s1, const string &s2)
```

Then we say the function `lexicoCompare`'s type is the TYPE of returning type of `int` and two parameters of type `const string&`. Denote by

```
int (const string &,  const string &)
```

2.3.2 Default Arguments

- A C++ function may have or may not have any arguments. This should be defined in the `.h` file or function prototype.
- In C++ a function/method can be given *default arguments*. Default arguments must be given to the rightmost parameters first. Because the number of arguments to a function with default values is variable, the default value parameters must appear after all parameters that don't have default values.
- Correct examples:

```
int sum(int = 0, int = 1);
void print(char *str = "");
```

- Incorrect example:

```
void readBytes(int fd = 0, void *buf);
```

2.3.3 Inline Functions/Methods

- Inline functions/methods can be defined either by preceding the function prototype with the `inline` keyword.
- Example 1:

```
inline int square(int x) {
  return x*x;
}
```

- Or placing a function definition inside a class definition, called an *inline member function*.
- Example 2:

```
class Card {
   public:
      face_t getFace() const
        { return face; }
      .
      .
      .
      face_t face;
};
```

- Inline member function has two effects:
  1. You don't have to write the function implementation later
  2. Each time the inline function is used in your program, compiler will recompile the short function definition and place a copy of this compiled short definition in your code.
- Notice that when you declare an inline member function, there is no semicolon before the opening curly bracket or after the closing curly bracket.

2.3.4 Implementing Member Functions

- Writing the complete definition of a member function is just like writing any other function, with one small difference: In the head of the function definition, the class name must appear before the function name, separated by two colons
- For example, if we want to define the member function `getFace()` of the class `Card` outside the body of the definition of the class `Card`, then we must write it as

```
face_t Card :: getFace( ) const
{
    return face;
}
```

- This requirement, called the *scope resolution operator*, tells the compiler that the function is a member function of a particular class.
- We use the term *function implementation* to describe a full function definition. The function implementation provides all the details of how the function works.
- Usually all the function implementation codes of a certain class are written in a separate file, called *implementation file* of the class while the prototype definition and all the information needed to use the class should appear in a *header file*.

2.4 Constant and Static Members

2.4.1 Constant Members

- Constant data members:
  - can be initialized only by constructor:
  - can not be changed once initialized.
- Constant methods must not:
  - alter the data members.
  - call other methods that are non-const.
  - call non-const methods of the data members.

2.4.2 Constant Examples

- Incorrect Example:

```
class foo {
   public:
     foo()
       { b = a = 0; }
     int x() const           //constant method
       { return y()*2; } //call a non-constant method
     int y()            //non-constant method
       { return a; }
     int *z()
       { return &b; }

   private:
     int a;
     const int b;
};
```

- Corrected Version:

```
class foo {
   public:
      foo(): b(0)
        { a = 0; }
      int x() const
        { return y( )*2; } //works well
```

```
        int y( ) const
          { return a; }
        const int *z( ) //return a pointer to int object
          { return &b; }      //which is of constant type

    private:
        int a;
        const int b;
};
```

2.4.3 Static members

- Any member (except constructors & destructors) can be declared *static*.
- Usually each object has its own copy of each member variable. However a static data member only has one instance (classwide). That is, all of the class's objects share the *same* value for a static data member.
- A static member is used w.r.t the whole class, not a specific object.
- Use the `static` keyword to declare a static member.
- Example:

```
class Card {
    public:
        static Card getAce()
          { return Card(ace, spades);}
    static Card getKing();
    static const int cardsInPack;
     ...
};
```

- In addition to declaring the static data members and methods within the class definition, the program must also repeat the declaration of the static members in the implementation file elsewhere. For the above example, we must declare the following stuff elsewhere

```
const int Card::cardsInPack = 52;

Card Card::getKing() {
  return Card(king, diamonds);
}
```

- For methods, `static` and `const` don't mix.

```
class X {
  static int foo() const;
  ...
};
```

where the member method `foo()` is declared as static and constant function.

- To use static data or methods, call with the class identifier other than a special object from the class, for example:

```
Card c = Card::getKing();
for (int i = 0; i < Card::cardsInPack; i ++)
  ...;
```

because we have only one copy of static members for all the objects of the class.

2.5 Operator Overloading

2.5.1 C++ Operators

- There are many operators in C++. They can be *binary operators* and *unitary operators*
- Binary operators take actions (operations) on two *operands*. Most of C++ operators are binary operators, for example, `+`, `-`, `==` and `>=` etc.
- Unitary operators take actions on a single operand. The typical unitary operators are `!` (negative boolean) and `-` (negative value).
- The primary type variable can be used as operands for C++ operators. For example, we can write

```
int a, b, c;
char letter = '=';
a = 1, b = 2;
c = a + b;
cout << 'c' << letter << c;
```

- However the following may result in an error

```
Card  A, B;
cout << A + B;
```

because C++ compiler does not know how to add two `Card` objects and how to output a `Card` object.

- Making operator available to new class objects is called *operator overloading*. For example, if we want to write `cout << A` where `A` is a `Card` object, we need to tell the compiler how to output the content of the objects for the `Card` class.

2.5.2 Overloading Operators as non-Member Functions

- Overloading an operator is just like defining a new function for a new class but the function name must take the format of the keyword `operator` followed by the overloaded operator.
- The general syntax for overloading a binary operator, using `+` as an example,

```
returnType operator + (const newClass& Op1, const newClass& Op2)
{
   //Algorithm definition
}
```

where the operator `+` can be replaced by any other binary operators such as `==` etc., and `Op1` and `Op2` are considered as the left operand and right operand, respectively. The type of the result of the operation may be different from the type of the operands.

- For example, consider the `vector` class for 3D space

```
class vector {
   public:
     vector();   //default constructor
     vector(int u, int v, int w) //constructor with three components
     {  x = u;
         y = v;
         z = w;   }
     ~vector(); //destructor
     ....;       //other member function
   private:
     int x;
     int y;
     int z;
}
```

- From mathematics we know the sum of two 3D vector is also a vector whose three components are the sum of the components of the two vectors respectively. Thus we can define `+` for the `vector` as

```
vector operator + (const vector& p1, const vector& p2)
{
  int sum_x, sum_y, sum_z;
  sum_x = p1.x + p2.x;
  sum_y = p1.y + p2.y;
  sum_z = p1.z + p2.z;
  vector sum(sum_x, sum_y, sum_z);
  return sum;
}
```

- In mathematics, another important operation is the inner product between two vectors. The result of inner product is the sum of the multiplications of the corresponding components. For two vectors `p1 = (x1, y1, z1)` and `p2 = (x2, y2, z2)` the inner product of `p1` and `p2` is

$$s = p1 * p2 = x1 * x2 + y1 * y2 + z1 * z2$$

The result is a scalar other than a vector. Hence the overloaded operator `*` should be defined as

```
int operator * (const vector& p1, const vector& p2)
{
  int product;
  product = p1.x * p2.x + p1.y * p2.y + p1.z * p2.z;
  return product;
}
```

2.5.3 Overloading Operators as Member Functions

- Sometimes overloading operator as a member function is convenient for users. For example, suppose both `x` and `y` are objects of a class, we hope that `x += y` has the same effect as `x = x + y` i.e., the operation result should be put back into `x`. In this case, it is better to define overloaded operator `+=` as a member function of the class.
- If a binary operator is overloaded as a member function of a class, then the object who calls this function will implicitly be the first operand of the operation with the other operand coming from the argument of the function. Hence the prototype for the overloaded operator as a member function looks like

```
returnType operator += (const newClass& RightHandOperand);
```

where only one argument in the parameter list of the overloaded function which is the right hand side operand.

- For example, let us define the `myString` class

```
class myString {
  public:
     myString(const char* = 0); //constructor
       ...
     myString& operator+=(const myString& rhs); //overload +=
       ...
  private:
     int _size;
     char *_string;
}
```

- Then define the function with new algorithms somewhere

```
myString& myString::operator += (const myString&  rightS){
   if (rightS) {
     myString tmp(*this);  //store the object itself
     _size += strlen(s);   //new size, this += for int
     delete[] _string;     //delete the storage for the object string
     _string = new char[_size + 1]; //new space for the concatenated string
     strcpy(_string, tmp._string); //copy the old string first;
     strcpy(_sring + tmp._size, rightS); //copy the string provided by s
     //Now we have finished the concatenation of the string with s
  }
  return *this;  //return new string
}
```

- Then in our program, we can write

```
mystring1 += mystring2; //concatenate string1 with string2
```

please read the above statement as the object `mystring1` is calling its member function `+=` with the argument `mystring2`.

2.5.4 Overloading Output and Input Operators

- The output operator `<<` and the input operator `>>` can be overloaded for a new class. However the overloading must follow the mysterious prototype shown below

```
ostream& operator << (ostream& outs, const newClass& source);
istream& operator >> (istream& ins, newClass& target);
```

- In the overloading prototype, the first argument and the return type must be an object of `ostream&` and `istream&`, respectively. Thus you *cannot* overload both operators as member functions of the new class.
- The `source` parameter is a `const` reference parameter, meaning that the function will not alter the point that it is writing.
- You need to provide the complete implementation for the overloaded output and input operators, that is, how to output and input. Our textbook provides examples for the class `point`.
- As both the functions for the overloaded output and input operators are not member functions of new classes, so both of them do not have access to the private members of the objects of the classes. To solve this problem, we can declare them as *friend functions* of the classes.

Chapter 3

The Bag Class

The bag class is one of special kinds of *container classes*. A container class is a class where each object contains a collection of items. The bag class that we will build is the simple version of more complex classess defined in the C++ Standard Library.

3.1 The Bags

3.1.1 Bags as Containers

- A container may hold a collection of items or elements. For example, a container containing several integers, a container containing several candy.
- Generally we make an assumption that the items in a container are of the same type.
- A bag is a special kind of container in which the items may be repeative, without any order.
- Our purpose is to implement the bag concept by using certain data structure.

3.1.2 Bag Classes and Data Members

- To implement any Bag collection, we need something to keep the bag elements in it. We will use a C++ class to implement the bag collection. In the class to be implemented an array will be used as the spaces for storing bag elements
- In C++, in order to declare an array, we need to specify the type of items of the array. To be more flexible, won't actually use a special type when we refer to the types of the items in the bag, i.e., in the array.
- We use the name `value_type` for the data type of the items in a bag. A `typedef` statement will be used to link a specified type to the type name `value_type`.
- To keep track of how many items are in a bag, we will use a variable. The type of the variable is defined as the name `size_type`. The name is normally specified to an integer type of some kind, for example, C++ data type `size_t` which is an integer data type that can hold only non-negative numbers.

- In the implementation, we may need to specify the size of the array in the Bag class, called the bag's *CAPACITY*.
- For example, our bag class for integers looks like

```
class bag
{
   public:
      typedef int value_type;
      typedef std::size_t size_type;
      static const size_type CAPACITY = 30;
      ....
}
```

Data member `CAPACITY` is declared as `static` and `const`, so all the instances from this class share a single copy of the `CAPACITY` and the value of `CAPACITY` is defined once and cannot be changed.

- In the above implementation, you change the `int` type to any other special types so that you can make the bag class for items of the types.

3.1.3 Member functions and Operations of Bag Classes

- The `bag` may have a default constructor to initialize a bag instance so that it is empty. Also we may need a copy constructor so that we can make a new object (or instance) of the `bag` class from another existed object, like

```
bag b;
//We put some items into the bag b here
bag c(b); //make a new bag copying from the bag b
```

- We may need a constant member function to count the number of the items contained in the bag. Once we call this function for an object of the bag, the number of all the items in the bag can be given. We call this function the `size()` function with the following prototype

```
size_type size() const;
```

- We definitely need methods to allow taking items out of the bag, declared as

```
bool erase_one(const value_type & target);
size_type erase(const value_type & target);
```

The `erase_one` function removes one copy of `target` and returns `true` or `false` depending on if the `target` is really in the bag. The `erase` function removes all the copies of `target` in the bag and returns the number of copies removed.

- We also need a method to allow putting an item in the bag.

```
void insert(const value_type & item);
```

- We need a method to report how many copies of a *particular* item are in the bag.
- Or we may need an external method to merge two bags, called the union operation which can be obtained by overloading the + operator. If the + operator is defined for bags, so it is sensible to also overload the += operator.

3.2 Class Design

3.2.1 Private Members and Constructors

- The class for the bag will have an array in it to hold the bag items. This array will be one of the private member variables of the `bag` class. The length of the array is determined by the constant `CAPACITY`.
- The entries of a bag will be stored in the front part of an array, as shown in this example. That is, the true number of items in the bag may be less than the constant `CAPACITY`.
- The entries may appear in any order. This represents the same bag as the previous one. This is the basic feature of the bag.
- Because part of the array can contain garbage, the `bag` class needs to keep track of how many numbers are in the bag. We put this information in the variable `used` of type `size_type`. One solution:

```
class bag
{
  public:
        ...
  private:
     value_type  data[CAPACITY];
     size_type used;
};
```

- Every method that is in the class will use this array, removing something from it or putting something in, etc. Remember that how you decide to represent a collection determines the inner workings of the class, the data structure type, including its variables and methods. These inner workings do not have to be known by the user of the class. The user can use the `bag` class without thinking about the array or whatever is being used. All the user of the class needs to know is what methods the class has and what parameters to send each method, and what each method returns.
- The default constructor initializes a bag as an empty bag, and does no other work. An empty bag does not have any items in it. That is, no array entry is used. So the variable `used = 0`

```
bag::bag() { used = 0; }
```

3.2.2 Counting

- The `size` function returns the number of items in the bag. This information is recorded in the private member `used`. So the function implementation is very simple

```
bag::size_type bag::size() const
{
   return used;
}
```

- To count the number of occurrences of a particular item `target` in a bag, we step through the used portion of the array, that is from `data[0]` to `data[used-1]`. The function definition is

```
bag::size_type bag::count(const value_type& target) const
{
   size_type answer;
   size_type i;
   answer = 0;
   for (i = 0; i < used; ++i )
      if (target == data[i] )
         ++answer;
   return answer;
}
```

3.2.3 Insertion and Deletion

- As there is no order among the bag items, so we can insert a new item into a bag at any place in the array. The simplest way is to put the first available position in the array, that is

data[used]. The only condition that should be checked is whether used==CAPACITY. If this condition is true, there is no available array entry for a new item. That is, the bag has been full. In this case we cannot insert a new item any more.

```
void bag::insert(const value_type& entry)
{
   assert(size() < CAPACITY ); // Check the condition
   data[used] = entry; //Put new item in the first available place
   ++used;  //Increase the number of the items
}
```

- The erase_one function removes an item named target from a bag. There may be several occurences of the target in the bag, however the erase_one function simply removes one occurrence. As the items of the bag are stored in the bag's array, in the first step the function looks for the target from the beginning of the array until the entry indexed by used-1. The first occurrence of the target along the array will be removed. To make the array partially filled, in the second step, the last item indexed by used-1 will be moved to the place where the first occurrence of the target stays.

  The following example demonstrates the whole deletion procedure. Suppose the target is the number 6. There are two occurences for the number 6. First the first number 6 is found in the second entry of the array, and it will be removed. Then the number in the fifty entry, i.e., 7, will be moved into the second place to make the array partially filled.

  If the target is successfully deleted, the target function returns a true boolean value, otherwise a false value in the case of no target in the bag.

```
bool bag::erase_one(const value_type& target)
{
   size_type index; //location of target in the data arry
   index = 0;
   while ((index<used) && (data[index]!=target))
      ++index;   //Search for target item by item
   if (index == used)
      return false;  //no target found
   --used;
   data[index] = data[used];
   return true;  //successful deletion
}
```

- With the aid of the erase_one function, it is easy to implement the erase function which deletes all the occurrences of a target from the bag if any.

```
bag::size_type bag::erase(const value_type & target);
{
   size_type number;
   number = 0;
   while (erase_one(target))
      ++number;
   return number;
}
```

3.2.4 Union Operations

- In our design, the operator `+=` is overloaded as a member function. The overloaded function has one parameter of type `bag` and copies each of items from the parameter bag to the object bag that activates `+=`.
- The core work of the overloaded function is to copy items from the array in one bag into the array of another bag. The Standard Library contains a `copy` function for easy copying of items from one location to another.

```
void bag::operator += (const bag& addend)
{
   assert(size()+addend.size() <= CAPACITY);
   copy(addend.data, addend.data+addend.used, data+used);
   used += addend.used;
}
```

3.3 General Rules

3.3.1 What need to learn?

- A container or a bag could be implemented in many ways. We just look at one implementation by using an array for the storage of items in the container.
- Although there are several built-in container classes in the C++ Standard Library, just using them does not give you a real understanding of the structure. So we choose to implement the bag container in our own way.
- Having to figure out how to handle the removal of an element in a container of a particular type makes you unable to avoid learning the container itself.

3.3.2 What to remember for each container?

- What are the characteristics of a container? For example, every container has to have a method to add new items and a method to delete items

- What are the characteristics that are peculiar to this particular container? For example, the bag container has no order, so that removal of each element should be random. Adding items can take place anywhere.
- What are the operations that must be defined for a container? This tells you what methods must be defined for the class implementing for the container.

Chapter 4

The Sequence Class

4.1 The Sequences

4.1.1 Specification

- A sequence is similar to a bag — both contain a bunch of items of the same typpe.
- But unlike a bag, the items in a sequence are arranged in an order, that is, you can follow the order to access the items one after another.
- In the implementation of bag classes, the items of a bag are arranged in an array, one after another. It seems that there is also an order in a bag. Please note, that is a quirk of our particular bag implementation, and it is not part of the natures of a bag. Actually there is no order for items of a bag at all.

4.1.2 Iterator

- The items of a sequence are kept one after another. The order is part of natures of a sequence.
- A program can step through the sequence one item at a time. A sequence class provides a program methods to control precisely where items are inserted and removed within the sequence.
- On a sequence a program can use an *internal iterator* to access items.

4.2 The Design of Sequence Classes

4.2.1 Data Types Used in Sequence Classes

- Like a bag class, our sequence is a class for a collection of items of an underlying `value_type`. Also our sequence provides `size_type` for the sequence size. Both type names will be declared by using `typedef` statement in the class.
- As we have done for defining a bag class, we use an array of size `CAPACITY` and type `value_type`. The constant `CAPACITY` is also declared in the class.

- For example, our bag class for double values looks like

```
class sequence {
   public:
      typedef double value_type;
      typedef std::size_t size_type;
      static const size_type CAPACITY = 30;
      ....
}
```

4.2.2 Data Members

- The class for the sequence will have an array in it to hold the sequence items. This array will be one of the private member variables of the `sequence` class. The length of the array is determined by the constant `CAPACITY`.
- The entries of a sequence will be stored in the front part of an array. That is, the true number of items in the sequence may be less than the constant `CAPACITY`.
- The entries must appear in sequence's order. This oder is one of the basic features of a sequence.
- Because part of the array can contain garbage, the `sequence` class needs to keep track of how many numbers are in the sequence. We put this information in the variable `used` of type `size_type`.
- In sequence class we define the so-called current item which indicates the precise location where we are. The current item is reprented by its index in the array, declared as `current_index`.
- One solution of the sequence class

```
class sequence {
  public:
        ...
  private:
     value_type  data[CAPACITY];
     size_type used;
     size_type current_index;
};
```

4.2.3 Member Functions and Operations

- A default constructor is needed for the sequence class. It builds up an empty sequence object.
- The `size` member function returns the number of items in the sequence.
- We will have member functions to exam a sequence which has already been built. For a bag, all information we can gather is how many copies of a particular item are in the bag. For a sequence, we may examine the items one after another and the items must be examined in order, from the front to the back of the sequence. Three member functions are provided by the sequence class to enforce the in-order retrieval rule. The `start` function moves to the front of the sequence and makes the front item as the current item. The `advance` function changes the current item to the next item in the sequence. The `current` function returns the information of the current item in the sequence.
- Generally the `current` function provides the information of the current item. However there exists a case where we cannot get the information of the current item. For example, if the current item is the last item in the sequence, then the current item does not exit after the `advance` function is called. So the sequence class provides a function, called `is_item`, to examine whether there actually is another item for `current` to provide, or whether `current` has advanced right off the end of the sequence.
- Like a bag class, the sequence class also provide member functions to insert new items into the sequence. The `insert` function places a new item before the current item while the `attach` function adds a new item to a sequence after the current item. The operation here is different from that of the insert function of a bag class where the new item is simply placed at the first available position in the array.
- We can remove the current item from a sequence by the `remove_current` function. This function has no parameters. In the `bag` class, the `erase_one` function has a parameter of `target` and one occurence of the target, if any, will be removed from the bag. For a sequence, we have no such a deletion function. If you really want to delete a particular item from a sequence, first you need to move the current position to the target then call `remove_current` function to remove the target item.
- You may add more member functions to a sequence class.

4.3 The Implementation of Sequence Classes

4.3.1 The Header File

- The header file for the sequence class

```
class sequence  {
    public:
        // TYPEDEFS and MEMBER CONSTANTS
```

```
        typedef double value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;
        // CONSTRUCTOR
        sequence( );
        // COPY CONSTRUCTOR
        sequence(const sequence& source)
        // MODIFICATION MEMBER FUNCTIONS
        void start( );
        void advance( );
        void insert(const value_type& entry);
        void attach(const value_type& entry);
        void remove_current( );
        // CONSTANT MEMBER FUNCTIONS
        size_type size( ) const;
        bool is_item( ) const;
        value_type current( ) const;
    private:
        value_type data[CAPACITY];
        size_type used;
        size_type current_index;
    };
```

- The function implementation is to be finished in Tutorial 2 (in Week 3).

4.3.2 The Function Implementation Examples

- All the function implementation should be completed in Tutorial 2. As an example, we only implement the `insert` function and the `current` function.
- In our `sequence` class declaration we have a user defined copy constructor. Its purpose is to create a new `sequence` object from another existed `sequence` object. The algorith is to copy every items in the existed object into the created object one by one. Here is the example of its implementation. You may have your own version.

```
sequence::sequence(const sequence& source)
// Library facilities used: cstdlib
{
     size_type i;
     used = source.used;
     current_index = source.current_index;
     for(i = 0; i < used; i++)
         data[i] = source.data[i];
```

```
}
```

- By our definition, the `insert` function places a new item before the current item. There are a lot of work to be done in the `insert` function. First of all we need to check if there exists any extra space in the entry for the new item. If the array is full, then you cannot insert any new items into the sequence.

  The current item is indexed by the private member `current_index`, so the new item should be put in the `data[current_index]`. If there is no current item, then the new item should be put in the front of the sequence. Whatever the case is, we need to move all the items from `data[current_index]` one place toward the end of the sequence so that we can make the space `data[current_index]` available to the new item and keep the order of the items in the sequence.

  The implementation code is as follows

```
void sequence::insert(const value_type& entry)
  // Library facilities used: cstdlib
  {
      size_type i;
      assert(size() < CAPACITY );
      if (!is_item())  //if there is no current item
          current_index = 0;
          // make the first item as the current one
      for(i = used; i > current_index; i--)
          data[i] = data[i-1];
          //move all the items after the current item
      data[current_index] = entry;  //put the new item
      used++;
  }
```

Chapter 5

Pointers and Dynamic Arrays

5.1 Pointers

5.1.1 Pointer Variables and Types

- The values belonging to pointer data types are the memory addresses of a computer. A pointer variable is a variable whose content is an address, or a memory location.
- When you declare a pointer variable, you need to specify the data type of the value to be stored in the memory location pointed to by the pointer variable.
- Please review Lecture 1 on how to declare a pointer variable. Because C++ does not automatically initialize variables, pointer variables must be initialized if you do not want them to point to anything. For example

```
int * p;
p = NULL; // or
p = 0;
```

5.1.2 Dynamic Memory

- Variables that are created during program execution are called *dynamic variables*. C++ provides two operators — `new` and `delete` (see Lecture 1) — to create and destroy, respectively, dynamic variables.

```
double * d_ptr;
d_ptr = new double;
```

- From the above example, you can see dynamic variables are not declared, even it does not have an identifier for a dynamic variable.

- Dynamic variables are created during the execution of a program. Only at that time does a dynamic variable come into existence.

5.1.3 Pointer Parameters

- A function parameter may be a pointer or an array.
- A value parameter of a function can be a pointer. An example was introduced in section 1.2.5 of Lecture 1.
- Sometimes it is possible to declare a parameter that is a reference to a pointer if we wish to modify the pointer itself rather than the object addressed by the pointer. For example, here is a function to swap two pointers

```
void ptrswap(int *& v1, int *& v2) {
    int * tmp = v2;
    v2 = v1;
    v1 = tmp;
}
```

  Can you tell what is the difference between this swapping function and the one defined in section 1.2.5 of Lecture 1.

5.2 Arrays and Array Pointers

5.2.1 Arrays

- An array is a collection of objects of a single data type. The individual objects are not named; rather, each one is accessed by its position inn the array.
- An array definition consists of a type specifier, an identifier (name for the array) and a dimension

```
int myArray[10];
```

- An array cannot be initialized with another array, nor can one array be assigned to another. Additionally, it is not permitted to declare an array of references.
- The array identifier evaluates to the address of the first element contained within it. Its type is that of pointer to the type of the element the array contains. For example, `myArray`'s type is `int*`.

5.2.2 Array Parameters

- Array could be a parameter of functions, called *array parameters*. An array parameter is passed as a pointer to its first element.
- To declare an array parameter in a function you can use any of three declarations formats, as shown in the following examples

```
void myFunc(int*);
void myFunc(int[]);
void myFunc(int[90]);
```

- As an array is passed as a pointer, thus the changes to an array parameter within the called function are actually made to the array argument itself and not to a local copy.
- The array's size is not part of its parameter type. The function being passed an array does not know its actual size, and neither does the compiler. Usually the size information is passed to a function by using an extra size parameter.

```
void myFunc(int* myArray, int Size);
```

- However if you can declare the parameter as a reference to an array. When the parameter is a reference to an array type, the array size becomes part of the parameter and argument types, and the compiler checks that the size of array argument matches the one specified in the function parameter type.

```
void myFunc(int (&myArray)[90]);
```

5.2.3 Using Dynamic Arrays

- An array created during the execution of a program is called a *dynamic array*. To create a dynamic array, we use the `new` operator to allocate an array of variables. Example

```
int *p;
p = new int[10];
*p = 25; // storing 25 into the first memory location
```

- In the above example, the pointer `p` is pointing to the first memory location of the allocated memory for the created array. Moreover, `p` is simply called a dynamic array.

- Any component in a dynamic array is accessed by index operator `[ ]`. For example, the third component in the dynamic array `p` is `p[2]`.
- A dynamic array must be destroyed by `delete []` operator. For example

```
delete [] p;
```

- For multidimensional array note that only the first dimension of the array dynamically allocated can be specified using an expression evaluated at run-time. The other dimension must be constant values known at compile time.

5.3 Function Pointers

5.3.1 The Type of Function

- In Comp131 and Comp132, you learnt a lot about C++ functions. For example, function prototype, function names, parameter list, and function definition etc.
- Like a variable, a function also has a type. A function's name is not part of its type.
- A function type is determined only by its return type and its parameter list.
- For example, consider the following function

```
int sizeCompare(const string & s1, const string & s2) {
  // function definition is here
}
```

where the function name is `sizeCompare`, and the *type* of the function is given by its parameter list consisting of two `const string &` parameters, and its return type `int`. Thus the following function shares the same type as the above

```
int yourComp(const string & m1, const string & m2) {
  // function definition is here
}
```

- In C++, you can define functions with the same names but in different types. The following example actually declares two "different" functions as compilers will perform function type checking before loading an appropriate function.

```
int sizeComp(string &, string &);
int sizeComp(const string &, const string &);
```

5.3.2 The Type of Member Functions

- A member function has an additional type attribute absent from a nonmember function, i.e., *its class*.
- A pointer to a member function must match the type of the function it is assigned, in three areas: (1) the type and number of parameters, (2) the return type, and (3) the class type of which it is a member.
- The declaration of a pointer to member function requires an expanded syntax that takes the class type into account. A pointer to a member function can be declared, initialized, and assigned, as follows

```
int (MyClass::*ptr_f1) () = NULL;
int (MyClass::*ptr_f2) () = &MyClass::AMemFun;
//where AMemFun is a member function with no parameters
//and a return type of int
ptr_f1 = ptr_f2;
ptr_f2 = &MyClass::AnotherMemFun;
//where AnotherMemFun is another member function with
//no parameters and a return type of int
```

5.3.3 Pointer to Functions

- For a function we can talk about its *function type*. Thus we can define pointer variables to function types.
- The syntax of defining a function pointer variable is given as follows

```
returnType (*F_PtrVar) (list of parameter types);
```

where `F_PtrVar` is the variable name being defined. The following example defines a function pointer variable named `myPtr`

```
int (* myPtr) (const string&, const string&);
```

Note: as a variable, `myPtr`'s type is a pointer pointing to functions of the type of two `const string&` parameters and a return type of `int`.

- Like an array name, a function name is considered as a function pointer (value) pointing to the function. Thus we can assign a function name (value) to a function pointer variable of the same type. For example, we can

```
myPtr = sizeCompare;
```

- It is possible to declare arrays of pointers to functions. For example

```
int (*testCases[10])();
```

declares `testCases` to be an array of ten elements. Each element is a pointer to a function that takes no arguments and that has a return type of `int`.

5.3.4 Function Pointer Parameters

- A function parameter can be a "pointer to function". That means we can pass a function argument to another function. We will use this technique in constructing sort algorithms.
- To see how to declare a "pointer to function" parameter, let's have a look at the following example

```
typedef int (*PFunc) (const string &, const string &);
int sortFunc(string *, string *, PFunc);
```

where the first line defines a name "`PFunc`" as the type of "pointer to function" pointing to functions of type of two `const string&` parameter and a return type of `int`.

- We can pass any function argument of type of two `const string&` parameter and a return type of `int` to the function `sortFunc`. See the example below

```
int lexicoCompare(const string &, const string &);
//the function lexicoCompare
string *s1, *s2;
//other stuff
sortFunc(s1, s2, lexicoCompare); //calling sortFunc
```

In this example, we are calling the function `sortFunc` with two `string` arguments `s1`, `s2` and a function argument `lexicoCompare` which itself is a function.

Chapter 6

Dynamic Implementation of Container Classes

6.1 The Bag Class with A Dynamic Array

6.1.1 Pointer Member Variables

- Pointers enable us to define data structures whose size is determined when a program is actually running rather than at *compilation time*. Such data structures are called *dynamic data structures*.
- A class may be a dynamic data structure (it may use dynamic memory) by using pointer member variables in the class.
- Several new factors come into play when a class has dynamic memory. The pointer member variables point to dynamic memory allocated at running time which are generally not considered as "part" of the class objects. Extra cares should be paid for dealing with the dynamic memory when the class objects are created and destroyed.
- The original `bag` class in lecture note 3 has a member variable that is a static array containing the bag's items. Now we will use a pointer member variable to create dynamic bag.
- The new bag class is roughly defined as

```
class bag {
   public:
      ...
   private:
      value_type *data; //Pointer to dynamic array
      size_type used; //How much of array is being used
      size_type capacity; //Current capacity of the bag
}
```

6.1.2 Constructor and Destructor

- The items of a bag object will be stored in a dynamic array that the member variable `data` points to. The constructor of the dynamic bag class is responsible for allocating the dynamic array.
- The initial size of the dynamic array is determined by the data member `capacity`. The size of the array may increase to whatever capacity is needed to all the items of the bag objects in a program.
- Increasing the size of the array ensures more items can be inserted.
- The prototype and definition of the constructor of the dynamic bag class may look like

```
bag::bag(size_type initial_capacity = DEFAULT_CAPACITY){
   data = new value_type[initial_capacity];
   capacity = initial_capacity;
   used = 0;
}
```

  where `DEFAULT_CAPACITY` is a default value defined elsewhere for the initial size of the dynamic array.

- As the dynamic memory for the bag items is allocated by the `new` operator in the constructor, the memory should be explicitly returned to the system by using the `delete` operator when the object goes out of scope. This can be done by the user-defined destructor.
- The primary responsibility of the destructor is releasing dynamic memory.

```
bag::~bag( ){
   delete [ ] data;
}
```

6.1.3 Copy Constructor and Assignment Operation

- In C++ when a new object is initialized/created as a copy of an existing object, the automatic (default) copy constructor will be activated. The automatic copy constructor copies data members one by one. The procedure is called *memberwise copy*.
- However if a class has pointer member variable such as the pointer `data` in our bag class, only can the pointer itself be copied from the existing object to the initialized object. The dynamic array pointed by `data` of the bag object won't be automatically copied.

- If you want to avoid the simple copying of member variables (in fact you must avoid this), then you must provide a copy constructor to do the job, copying the contents in the dynamic array.

- The copy constructor of the dynamic bag class should have the following prototype and implementation

```
bag::bag(const bag& source){
   data = new value_type[source.capacity];
   capacity = source.capacity;
   used = source.used;
   copy(source.data, source.data+used, data);
}
```

  the parameter of the copy constructor is usually a `const` reference parameter.

- The copy constructor is activated when you declare a new object by

```
//suppose x is an existing object
bag y(x);   //declare the object y
```

  or

```
bag y = x; //Alternative format of the above
```

- Like the automatic copy constructor, the automatic assignment operator makes a memberwise copy between the objects of a class. If a class, like our dynamic bag class, contains any pointer variable members, we must overload the assignment operator to make sure the contents in the dynamic array to be correctly copied.

- When you overload the assignment operator, C++ requires the overloaded operator to be a member function of the class. In an assignment statement `y=x` (both `x` and `y` are objects of the same class), the object `y` is activating the function, and the object `x` is the argument for the parameter of the function.

- The prototype and implementation of the overloaded assignment operator looks like

```
void operator =(const bag& source) {
    value_type *new_data;
```

```
        if (this == &source)  //self-assignment checking
            return;
        if (capacity != source.capacity) {
            new_data = new value_type[source.capacity];
            delete [] data;
            data = new_data;
            capacity = source.capacity;
        }
        used = source.used;
        copy(source.data, source.data+used, data);
    }
```

- As a general rule we always use the self-assignment checking to avoid the assignment such as `y = y` from occurring.

6.1.4 Other Member Functions

- Just like the static implementation of the bag class introduced in Lecture Note 3 we may have the following member functions.
- We may need a constant member function to count the number of the items contained in the bag. Once we call this function for an object of the bag, the number of all the items in the bag can be given.
- We definitely need methods to allow taking items out of the bag. The `erase_one` function removes one copy of `target` and returns `true` or `false` depending on if the `target` is really in the bag. The `erase` function removes all the copies of `target` in the bag and returns the number of copies removed.
- We also need a method to allow inserting an item in the bag.
- We need a method to report how many copies of a *particular* item are in the bag.
- Or we may need an external method to merge two bags, called the union operation which can be obtained by overloading the + operator. If the + operator is defined for bags, so it is sensible to also overload the += operator.
- The += operator is defined as

```
void bag::operator +=(const bag& addend) {
  if (used + addend.used > capacity)
     reserve(used+addend.used); //Allocate more memory
  copy(addend.data, addend.data+addend.used, data+used);
  used += addend.used;
```

```
}
```

- The + operator (as an external function) is defined as

```
bag operator +(const bag& b1, const bag& b2){
   bag answer(b1.size() + b2.size());
   answer += b1;
   answer += b2;
   return answer;
}
```

6.2 General Rules for Class with Pointer Member

6.2.1 Rules

- You may need member functions to allocate and release dynamic memory as needed.
- You must override the automatic copy constructor and the automatic assignment operator.
- The class must have a destructor to return all dynamic memory allocated to the object from the heap.
- Overall when a member variable of a class is a pointer to dynamic memory, the class should always be given a destructor, an overloaded copy constructor, and an overloaded assignment operator

6.2.2 When the Copy Constructor Used

- The copy constructor is used when one object is to be initialized as a copy of another existing object, as

```
//suppose x is an existing object
bag y(x);    //declare the object y
```

or

```
bag y = x; //Alternative format of the above
```

- When a return value of a function is an object of the class, by default the copying occurs by using the automatics copy constructor, which copies all the member variables fro the local variable to the return location. If you want to avoid the simple copying of member variable, then you must provide your own copy constructor.
- When a function uses a value parameter of an object of the class, the actual argument is copied to the formal parameter. If there is any pointer member, you need to provide your own copy constructor.

Chapter 7

Linked Lists and Their Applications

In this lecture, we will talk about the linked lists, and discuss the dynamic implementation for the linked lists.

7.1 Lists

7.1.1 Basic Definition

- A *list* is a linear data structure. We can *insert* and *delete* elements in any order from the list.
- A *linked list* is a new data structure which is used to implement a list of elements arranged in some kind of order. It is a sequence of elements arranged one after another, with each element connected to the next by a link.
- A *node* is a container for data in a list. In the list, each node may have a *successor* and a *predecessor*, but no explicit order. Each node knows only its successor and predecessor if any.
- Operations can take place anywhere inside the list.
- There are two implementations, *Dynamic implementation* and *Static implementation*

7.1.2 Node Classes

- Each node needs two bits of data: a piece of data information (any class type) and a link to the next node
- The node class can be defined as

```
class node {
  ...
  private:
     data_type data_field;
     node * link_field;
```

```
}
```

where *data_type* may be any valid data type such as the primary types (`int, double`) or user-defined classes.

- For the `node` class we would like to provide members for the node constructor, for setting data information and link, and for getting (reading) data information and link. For example, a simplified version of the node class is

```
class node {
  public:
     node(const value_type& init_data = A_Default,
        node* init_link = NULL) {
        data_field = init_data;
        link_field = init_link;
     }
     void set_data(const value_type& new_data) {
         data_field = new_data;
     }
     void set_link(node* new_link) {
         link_field = new_link;
     }
     value_type data() const
      {   return data_field;   }
     node* link()  { return link_field; }
  private:
      data_type data_field;
     node * link_field;
}
```

7.2 General Linked Lists

7.2.1 Building Linked Lists

- In the implementation of a linked list, the list is accessed through one or more *pointers* to nodes. That is, the linked list consists of several nodes linked one by one.
- A pointer to the first node of the list is called the *head pointer*. And the pointer to the last node of the list is called the *tail pointer*.
- We could also implement and maintain pointers to other nodes in a linked list. For example, we can define a head pointer, a current pointer to the so-called *current node* and a pointer to the previous node of the current node.

- Each pointer to a node must be declared as a pointer variable. In our implementation we won't declare a linked list class but instead we declare two pointer variables

```
node* head_ptr;
node* tail_ptr;
```

- The implementation of a linked list is defined and manipulated by a linked-list toolkit although we can declare a new class for the linked list data structure.
- A empty linked list means the value in the pointer variable `head_ptr` is `NULL`. And the `link_field` of the node pointed by the variable `tail_ptr` is always `NULL` (why?).

7.2.2 Functions in A Linked-list Toolkit

- One of the most imporatant methods is to insert a new node to the linked list. There are two cases to be handled.
  - Inserting a new node at the head of a linked list. The work to be done in this insertion process is to allocate new place (a new node) for the new element, to put the new node at the head of the linked list and to update the variable `head_ptr` of the linked list object. The prototype and implementation is as follows

    ```
    void list_head_insert(node*& head_ptr, const value_type& entry) {
      head_ptr = new node(entry, head_ptr);
    }
    ```

    The node constructor will allocate memory for the entry and set the `link_field` of the newly allocated node to the head node (pointed by `head_ptr`) of the list and the member function sets the `head_ptr` to the new node. That is, we have added the new node at the head of the list.
  - Inserting a new node at a place rather than the head of a linked list. In this insertion, we need to know where the new entry is to be added. We assume that a particular node is pointed by a node pointer `previous_ptr` and we will add the new entry just after that node. The work to be done includes allocating a new node (to be done by the node constructor), placing the new entry in the data field, making the link field to the node after the new node's location and connecting the `link_field` of the node pointed to by `previous_ptr` to the new node that we just created.

    ```
    void list_insert(node* previous_ptr, const value_type& entry) {
      node* insert_ptr;
      insert_ptr = new node;
      insert_ptr->set_data(entry);
    ```

```
    insert_ptr->set_link(previous_ptr->link() );
    previous_ptr->set_link(insert_ptr);
  }
```

- Searching is also an important task in manipulating linked lists. This operation find the node in the linked list by comparing the data field of the node with a given data entry and the function returns a pointer to the found node. The prototype of the function is

  ```
  node* list_search(node* head_ptr, const value_type& target);
  ```

  The searching process begins from the head node of the linked list and search for the target node by node until the target is found or not.

- Sometimes it is very important to remove a certain element from the linke list. Just like inserting an element into a linked list, we define two different functions for removing the head node and any other node, respectively. We need to provide the location of the node in the linked list. Two function take the following formats respectively

  ```
  void list_head_remove(node*& head_ptr);
  void list_remove(node* previous_ptr);
  ```

  where `previous_ptr` is the pointer to the node just before the node to be removed. In other word, the node pointed to by `previous_ptr` is in the front of the node to be removed. Here is the example of its implementation

  ```
  void list_remove(node* previous_ptr) {
    node * remove_ptr;
    remove_ptr = previous_ptr->link();
    previous_ptr->set_link( remove_ptr->link() );
    delete remove_ptr;
  }
  ```

  Question: can we use the above function as

  ```
  list_remove( tail_ptr );
  ```

- More member functions can be found from the textbook.

7.2.3 The Whole Linked List Toolkit

```
class node {
  public:
    node(const value_type& init_data=value_type(),
       node* init_link = NULL)
    { data_field = init_data;
       link_field = init_link; }
    node* link() { return link_field; }
    void set_data(const value_type& new_data){
       data_field = new_data; }
    void set_link(node* new_link) {
       link_field = new_link; }
    value_type data() const {
       return data_field; }
    const node* link() const {
       return link_field; }
  private:
    value_type data_field;
    node* link_field;
}
//Functions for the linked list
std::size_t list_length(const node* head_ptr);
void list_head_insert(node*& head_ptr, const value_type& entry);
void list_insert(node* previous_ptr, const value_type& entry);
node* list_search(node* head_ptr, const value_type& target);
const node* list_search(const node* head_ptr,
     const value_type& target);
node* list_locate(node* head_ptr, std::size_t position);
const node* list_locate(const node* head_ptr,
     std::size_t position);
void list_head_remove(node*& head_ptr);
void list_remove(node* previous_ptr);
void list_clear(node*& head_ptr);
void list_copy(const node* source_ptr, node*& head_ptr,
   node*& tail_ptr);
```

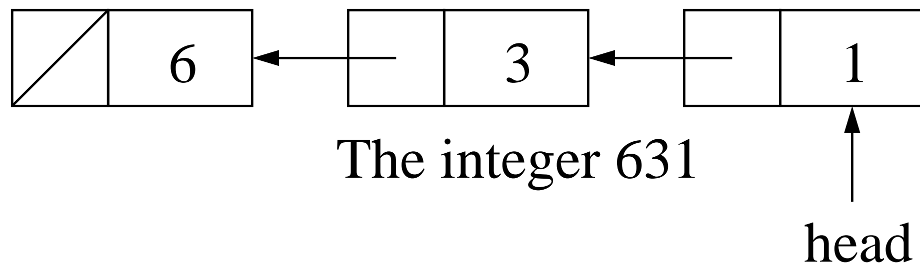
7.3 Application: Large Integers & Lists

7.3.1 List for Larger Integers

- Maximum integer using:

- *int* type: 2147483647
- *unsigned int* type: 4294967295

- Problem: What if we need integers larger than this?
- Solution: Use a linked list of digits, to create arbitrarily large integers.



- Example:

7.3.2 Pointer addition

- Given a pointer: `type *p`
- Assume `p` points to an array in memory
- `*p` is the first element
- `*(p+1)` is the second element
- `*(p+n)` is the (n+1)th element
- When we add `1` to a pointer, we actually add `sizeof(type)` bytes to the memory location

7.3.3 Initializing from a string

- We use a list of `value_type` being `short` to store the integer
- Function, initialize list `num` from string `str`:

```
void init(node* &num, char *str) {
  num = NULL;
  list_head_insert(num,(*str)-'0');
  str = str + 1;
  while ((*str) != '\0') {
    list_insert(num, (*str)-'0');
    str = str + 1;
  }
}
```

7.3.4 Adding two integer lists

- We can add the lists as we would on a piece of paper

  Digit by digit

- Basic Idea:

  - Traverse the two lists simultaneously
  - sum the current digits & add to new list
  - Remember *carry*

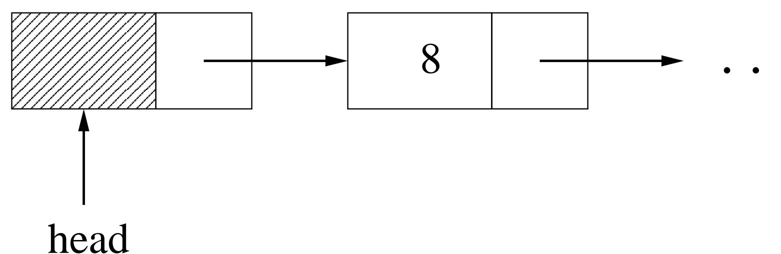
Chapter 8

More About Linked Lists

8.1 Other Techniques for Lists

8.1.1 Dummy First Node

- Dummy first node comes before the logical first node
- Doesn't contain any meaningful information
- Advantage
  - can avoid special cases
    - insertion & deletion of logical first node
    - insertion into empty list
- Example:



8.1.2 Circular Lists

- Eg: consider days of week
  - Monday, first day in week
  - Sunday, last day in week
  - Sunday is predecessor of Monday

- a circular list is:

  A list where the *head* is the successor of the *tail*

- Advantage: Starting from any node, we can traverse the list to any other node.

8.1.3 Doubly Linked Lists (*pp. 261 - 263*)

- In certain applications, we must be able to traverse the list in both directions
- Each node has a pointer to it's previous node.
- We can include a method to go back to the previous node in lists
- A possible set of definitions for a doubly linked list of items is the following

```
class dnode {
   public:
      ... ...
   private:
      value_type data_field;
      dnode* link_fore;
      dnode* link_back;
};
```

- For the head node

```
link_back = NULL;
```

- For the tail node

```
link_fore = NULL;
```

8.2 One More Example: Binary Encoding

8.2.1 Definition

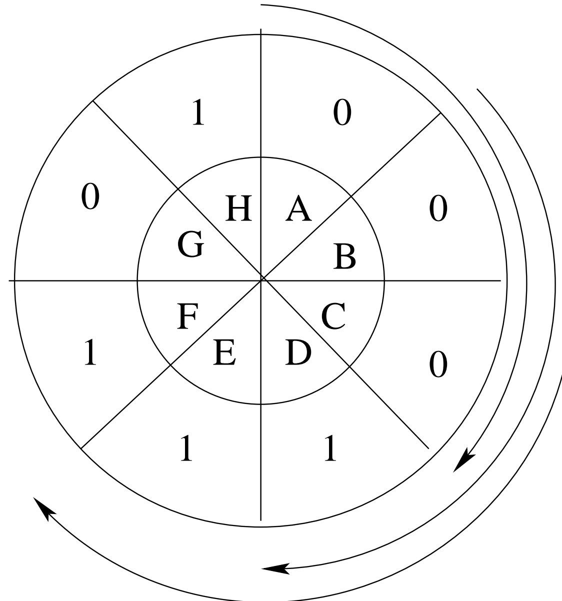
- A *deBruijn sequence* is a sequence of 2^n bits such that when it is arranged in a circle, each subsequence of n bits is different.
- Example: (length $2^3 = 8$)

```
00011101
```

- Can encode 8 characters, with 3 bits for each character
- Each 3 bit subsequence is a different character

```
000  A   001  B   011  C   111  D
110  E   101  F   010  G   100 H
```

- deBruijn sequences can be implemented as circular lists. See the following figure



- The data field of the node in the list consists of a bit (1 or 0) and a char.

8.2.2 Basic algorithm for encoding string

- A deBruijn sequence and chars represented by the sequence can be stored in a doubly linked list. The list can be built up from a file containing characters and another file containing deBruijn sequence.
- Suppose a doubly linked list of a deBruijn sequence has been given, then we can design a program to encode a text file.
- The algorithm is simple. Read each char from the text file, look for the char in the doubly linked list, read n consecutive bits from the list and write down the code for the char until no char remained in the text file.

```
while more chars to encode
  get next char
  traverse list until we reach that char
  output bits at next n nodes
end while
```

Chapter 9

Templates

In this lecture, we study the concept of template class and show several examples of how to use template classes and template functions.

9.1 Template Functions

9.1.1 Why Template Functions?

- Template function technique is a more flexible mechanism which provides function definition in terms of some underlying data type.
- We can instantiate many functions (following the same algorithm but for different types of data) from a template function.
- Actually a template function is not an ordinary function but just a template for instantiating new functions which apply same task to arguments of different types.

9.1.2 Syntax for a Template Function

- In a template function, there are three types of arguments
  - Type parameters
  - Non-Type parameters
  - Template parameters
- The general syntax to define a template function is

```
template <class T1, class T2, ...
          type1 v1, type2 v2, ...
          template_def1, template_def2, ...>
//The above is called template prefix
ret_type Func_Name(An_Arg_list) {
```

```
    ...
  }
```

- Swap function Example:

```
template <class T>
void swap(T *x, T *y) {
  T temp = *x;
  *x = *y;
  *y = temp;
}
```

9.1.3 Using a Template Function

- A program can use a template function with any `T` type that has the necessary features.
- In the case of the above `swap` template function, the type parameter `T` can be any of the C++ build-in type, or it may be a class. That is, we can use the above template function to swap two elements of any type.
- For example, we can use it as

```
swap(integer1, integer2); //swap two int variables
swap(s1, s2); //swap two string variables
swap(student1, student2); // swap two variable of
                          //two student objects
```

9.2 Template Classes

As we have seen, the function template technique can be used to define functions fitting to different cases where functions' arguments may be of different types. As a class is usually made of data fields with other types, one may use the template techniques to class declarations.

9.2.1 Idea for Template Classes

- A *template class* is a prescription for creating a sort of classes where one or more types or values are parameterized.
- Template classes use the same parameters as a function template, that is
  - Type parameters
  - Non-type parameters
  - Template parameters (the parameter is itself a template class).
- Template classes are useful when we want a same basic class structure, for an unlimited number of possible classes.

- They are particularly suited to structures that store different kind of things.
- Templates provide the means with which programs can build their own generic containers (class) and provide a mechanism for data abstraction.

9.2.2 Definition of Template Classes

- Syntax for a template class is

```
template <class T1, class T2, ...,
          type1 v1, type2 v2, ...,
          template_def1, template_def2, ...>
class Name {
  ...
  class definition
  ...
};
```

- The expression `template < ...   ...>` is called the template prefix. It warns the compiler that the following definition will use an unspecified data types, such as `T1`, `T2` etc.
- Each `T1, T2` etc. can be used as a type within the template class definition.
- Each `v1,v2` etc. are normal variables throughout the template class definition.
- Example:

```
template <class T, int n>
class foo {
  ...
  T  x[n];
  void replace(const T& newElement);
};
```

  This example is very interesting. Template class `foo` has a type parameter `T` and a non-type parameter `n` of the type `int`.

  In the body of the class definition, an array of type `T` (the actual type value is unspecified), `x`, is defined and the length of the array `x` is `n` which is also unspecified.

  From this template class one may build up a class including an array of any given length where the type of the array elements can be specified according to applications.

- `template_def` is of the form:

```
template <...> class ident
```

- Template Classes of arbitrary complexity can be created

```
template <template <class X, int a> class T>
class Y {
  T<int, 10> foo;
  ...
};
```

where the template class `Y` is defined in terms of another template class `T` with a type parameter `X` and a non-type parameter `a`, and in the definition of `Y` a specified/generated class `T<int, 10>` from its template is used.

9.2.3 Defining the methods for template classes

- A template class may contain many member function declarations whose definitions depend on the unknown type parameters etc.
- For the function implementation, you must follow some rules to tell the compiler about the dependency on the type parameters etc. Thus each method defined outside the class definition and each static data member must be also prefixed by the template keyword and template parameters.
- The function definition outside the class template body looks much like the syntax for template functions. For example, the *replace* function in the *foo* template class should be defined outside the template class body in this way

```
template <class T, int n>
void foo<T, n>::replace(const T& newElement){
  //replace operations here;
}
```

- For methods which use extra template arguments, we need to include *template* keyword within the class definition. This is also true for normal classes with method templates.

```
class Foo {
  template <class T>
    void fooThis(T &x);
  ...
};
```

This way means that we are using a template function within a template class or a normal class.

9.3 Applying Template Classes

9.3.1 Template Instantiation

- A template class is not a class at all, but just a plan to make new classes with the same structure.
- Generation of classes from a template class is called a *template instantiation*.
- A template instantiation can be used wherever a non-template class can be used.
- Arguments for template classes never be deduced from context.
- Before we can *instantiate* a template class, its definition, not declaration, must be known.

```
// declaration of X
template <class T>
class X;

// definition of Y
template <class T>
class Y { ... };

X<int> x;  // Error: only declaration given
Y<char> y; // OK. definition known
```

9.3.2 Assumptions About Type Parameters

- At times it is necessary for us to make assumptions about the type parameters of a template class or function. We may need to assume that a certain method is present in the class.
- For example, if we were writing a generic function for sorting an array of type `T`. We would assume that two values of type `T` could be compared.
- Fortunately C++ lets us make any assumptions about the type arguments we want. If we instantiate the template with arguments which make any of the assumptions false, the compiler will complain.
- Example:

```
template <class T>
class X {
public:
  X() {t(10, "foo");} // assumption: T has a constructor
                      //             T::T(int, char*)

  void bar()
  { t.doBar();       // assumption: T has a method doBar
  t.print(); }       // assumption: T has a method print
```

```
private:
  T t;
};

class Y {
public:
  Y();
  Y(int, char*);

  void doBar();
  void print();
};

X<Y> xy;     // OK. all assumptions about Y are true
X<int> xint; // Error: Assumptions are false
```

Do you know why the second instantiation does not work through?

9.3.3 Default arguments

- Like default arguments for function parameters, the formal parameters of template could have default values.
- The default values must be given to rightmost parameter first.
- If all parameters are default, must give empty brackets <>.
- Example:

```
template<class T = int>
class foo {
  ...
};

foo<> x; // Correct default int used
foo   y; // Incorrect, missing <>
```

Do you know why missing <> causes trouble?

9.3.4 Arguments for non-type parameters

- The parameter values to the non-type parameter must be a constant expression.
- And they must be determinable at compile time.

```
template <int x, int y>
class Pic { ... };

const int i = 4, j = 10;
Pic<i, j> p1;   // OK. i & j are const
Pic<3, 11> p2;  // OK.

void main() {
  int a, b;
  cin >> a >> b;
  Pic<a, b> p3; // Error: a & b not const
}
```

The compiler doesn't know the values for `a` and `b` at the compile time.

9.3.5 Static Data Members in Class Template

- Class templates can declare static data members.
- One set of static data members per instantiation. Normally there is only one copy of static data members for all the objects of a class. One can make many classes from a class template. If there are static data members defined in a class template, then each newly created class from the template has its own copy of static data members when instantiating.

Chapter 10

Templates and Iterators

10.1 Node Template Classes

10.1.1 Definition

- Node is a container for storing data of certain type together with a link to another node. In the former lectures, we use `value_type` as an alias for the type of data stored in the node.
- By using template class, we will replace the alias `value_type` by a type parameter `Item`.
- All member functions will be re-defined following the syntax of template classes.

10.1.2 New Node Template Class

- The prototype of the node template class is as follows

```
template <class Item>
class node {
  public:
     node(const Item& init_data=Item(), node* init_link=NULL)
     { data_field = init_data;  link_field = init_link; }
     Item& data() { return data_field; }
     node* link() { return link_field; }
     void set_data(const Item& new_data){
          data_field = new_data;  }
     void set_link(node* new_link){
          link_field = new_link;  }
  private:
     Item data_field;
     node* link_field;
};
```

- From the definition we can see that the `data_field` is of type `Item` which is currently unknown and should be instantiated later.

10.1.3 Toolkit for Linked Lists

- The toolkit for manipulating linked lists based on the node template class consists of a range of template functions.
- Here are several typical functions in the toolkit,

```
template <class Item>
void list_clear(node<Item>*& head_ptr);

template <class Item>
void list_copy(const node<Item>* source_ptr, node<Item>*& head_ptr,
     node<Item>*& tail_ptr);

template <class Item>
void list_insert(node<Item>* previous_ptr, const Item& entry);

template <class Item>
std::size_t list_length(const node<Item>* head_ptr);
```

- Please read the text (pages 306-313) for more details.

10.2 STL and Their Iterators

The ANSI/ISO C++ Standard provides a variety of container classes called the *Standard Template Library* (STL).

10.2.1 The Multiset Template Class

- A multiset is an STL template class similar to our bag. It permits a collection of items to be stored, where each item may occur multiple times in the multiset.
- The declaration of multiset is included in the standard header file `set`.
- Here is an example to declare a multiset (bag) object `mySet` containing `int` items

```
multiset<int>  mySet;
```

- The above statement invokes the default constructor creating an empty multiset object `mySet`.

10.2.2 Multiset Operations

- Multiset template class provides a lot of member functions which can be used to manipulate the set, such as inserting items into the set, deleting items from the set, counting the number of items in the set etc. For example, insert the integer 4 into `mySet`

```
mySet.insert(4);
```

- Other functions are

`count(target)`	returns the number of times that target occurs
`empty()`	returns true if the set has no items
`erase(el)`	removes all copies of el
`clear()`	removes all elememnts from the set

10.2.3 Iterators

- The multiset's `insert` function is different from the `insert` function that we defined for our bag class. The multiset insertion function returns a special value called an `iterator`, however the bag's insertion function returns a `void` type.
- An `iterator` is an object that permits a programmer to easily step through all the items in a container, examining the items and perhaps changing them.
- Any STL container has a standard member function called `begin` which returns an iterator that provides access to the first item in the container. Remember we have no order among the elements in a container (e.g. a set here), but the STL maintains ways in which elements are stored in certain order. Then the `begin` function gives the iterator to the first element. For example

```
multiset<string> actors;
//insert information into actors

multiset<string>::iterator myIndex;
myIndex = actors.begin();  //the iterator to the first actor
```

- You can imagine that an iterator of an element is the "index" to the element. Through the iterator you can get an access to the element. For example, suppose `myRole` is an iterator to the `actors` set, then `*myRole` is the item that the iterator `myRole` is "pointing" to. That is, an iterator has much meaning of an ordinary pointer.
- When you use the `++` to an iterator, the iterator will be moved to the "next" item in the container. This is the easy way to traverse all the elements in the container. For example, `++myRole` is a new iterator "pointing to" to the next actor in the set `actors`.

- A container also has an `end` member function that returns an iterator to mark the end of its items in the container. The typical way to traverse all the elements in `actors` is as follows

```
for (myRole = actors.begin(); myRole!=actors.end(); ++myRole)
{
  //do what you want to do here!
}
```

- Be careful: When an iterator's underlying container changes, for example, by an insertion or a deletion, the iterator generally becomes invalid. You need to reassign a new value to your iterator from the changed container.

- All iterators in STL can be grouped into three categories
  - *Forward Iterator*
  - *Bidirectional Iterator*
  - *Random Access Iterator*

10.3 Node Iterator — Linked List

The objective here is to construct node iterator for our linked lists.

10.3.1 Node Iterator

- We hope out node iterator allows us to step through the list. The proposed node iterator template class is derived from `std::iterator`.

- The prototype of the template class is

```
template <class Item>
class node_iterator
: public std::iterator<std::forward_iterator_tag, Item>
{
   public:
      //iterator methods
   private:
      node<Item>* current;
}
```

- From the definition we can see there is only one private member in the `node_iterator` template class which is a pointer to `node<Item>`.

10.3.2 Operations for Node Iterator

- The node iterator has one constructor with one parameter. The parameter is a pointer too a node where the iterator will start.

```
node_iterator(node<Item>* initial = NULL) {
    current = initial;  }
```

- We will overload the `*` operator for the aim of getting information of the item "pointed to" by the iterator

```
Item& operator*() const
{ return current->data();  }
```

- Also we should define `++` operations for our iterator, i.e., the prefix and postfix `++` operators

```
node_iterator& operator ++() //Prefix
{   current = current->link;
     return *this;    }

node_iterator operator ++(int) //Postfix
{   node_iterator original(current);
     current = current->link;
     return original;    }
```

- Finally we may need comparison operations for our iterators

```
bool operator ==(const node_iterator other) const
{  return  current == other.current;  }
bool operator !=(const node_iterator other) const
{  return  current != other.current;  }
```

Chapter 11

Complexity Analysis of Algorithms

The context of this lecture is from *Section 1.2* of the textbook. The **big-oh** technique is used to analyze the effectiveness of algorithms. We provide many examples to demonstrate the application of big-oh.

11.1 big-oh

11.1.1 Time and Space Complexity

In doing a time analysis for an algorithm, we count certain operations that occur while carrying out an algorithm or a method other than count the the actual elapsed time during each algorithm. That is, we do not usually measure the actual time taken to run the algorithm because the number of seconds can depend on too many extraneous factors — such as the speed of the processor, and whether the processor is busy with other tasks.

- There is no precise definition of what constitutes an *operation*. An operation can be as simple as the execution of a single program statement
- In computing the efficiency of an algorithm, some measure of the amount of work done or the space requirements must be used. This measure is called the *complexity* or *order of magnitude* of the algorithm.
- For most algorithms/programs, the number of operations depends on the algorithm /program's input. The complexity of an algorithm is measured in terms of the size of the problem/input using an expression of size n. [Those of you who have done *amth140* should know what I'm talking about].
- The method that we will use is to perform upper bound estimates of such as the complexity and efficiency.

11.1.2 An Example

- A polynomial $P_n(x)$ of order n is defined by

$$P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$$

- We want to evaluate polynomial $P_n(x)$ at a given point x. Two algorithms are available
- Algorithm A (as defined):

$$P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$$

- Algorithm B (more sophisticated):

$$P_n(x) = (((a_0x + a_1)x + a_2)x + \cdots + a_{n-1})x + a_n$$

- Now we want to analyze the number of multiplications used in both algorithms (we don't care about addition as it is much easier than multiplication on a computer)
- $\frac{n(n+1)}{2} = 0.5n^2 + 0.5n$ multiplications in algorithm A, and only n multiplications in algorithm B
- You may note that Algorithm B is faster than Algorithm A especially for larger n.
- If we use both algorithm to a new polynomial with order twice n, that is, $P_{2n}(x)$, then the number of multiplication of algorithm A increase about 4-fold while algorithm B twice fold. If we increase the order of polynomial to $10n$, then algorithm A need about 100-fold multiplication while algorithm B tenfold.
- The critical amount for Algorithm A is n^2 (not its coefficient 0.5) and the critical amount for Algorithm B is n. We can express this kind of information in a format called *big-O notation*. For example, we say the complexity of algorithm A is $O(n^2)$ (we don't care about the coefficient 0.5 and the second term $0.5n$) and algorithm B $O(n)$

11.1.3 Definition of big-O

- Function f is of *complexity* or *order at most* g, written with **big-oh** notation as $\mathbf{f} = \mathbf{O}(\mathbf{g})$, if there exists a positive constant c and a positive integer n_0 such that

  $$\mid f(n) \mid \leq C \mid g(n) \mid$$

  for all $n \geq n_0$. We also say f has complexity $O(g)$.
- The values C and n_0 in the above definition themselves are not important, the significance lies in the existence of such C and n_0, rather than the magnitude of these values.
- For example, the complexity of Algorithm A in the last subsection satisfies

$$|0.5n^2 + 0.5n| \leq 0.6|n^2|$$

  with $C = 0.6$ and $n \geq n_0 = 3$

Table 11.1: The number of Operations/time in Evaluating A Polynomial

Input Size (n)	Algorithm A ($0.5(n^2 + n) = O(n^2)$)	Algorithm B ($n = O(n)$)
10	55	10
100	5050	100
1000	500500	1000
10000	50005000	10000
100000	5000050000	100000
...	...	...

- See the `amth140` webpage for information on **big-oh**:

    `http://mcs.une.edu.au/~amth140`

11.1.4 Equality and Inequality of complexity

- f and g are of *equal complexity*, written $\mathbf{O(f)} = \mathbf{O(g)}$ provided $f = O(g)$ and $g = O(f)$.
- f is of *smaller complexity* than g, written $\mathbf{O(f)} < \mathbf{O(g)}$, if $f = O(g)$ but $g \neq O(f)$.
- For example, Algorithm B ($O(n)$) has a smaller complexity than Algorithm A ($O(n^2)$). Why? Because $O(n) = O(n^2)$ but $O(n^2) \neq O(n)$ (i.e., you cannot find a constant C and an integer n_0 such that $|n^2| \leq C|n|$ for all $n \geq n_0$)
- Generally, for a problem of size n, suppose that there are two algorithms A and B.
- Execute in $f(n)$ and $g(n)$ amounts of operation/space for algorithms A and B, respectively.
- If $\mathbf{O(f)} < \mathbf{O(g)}$, then, for *large enough* values of n, A will be more efficient than B.
- The following table shows you why an $O(n^2)$ algorithm is much worse than an $O(n)$ algorithm

11.2 Main Properties of big-oh

11.2.1 General Principles

- $f(n) = O(f(n))$
- $k \cdot f(n) = O(f(n))$
- $g(n) = O(f(n)) \rightarrow f(n) + g(n) = O(f(n))$

11.2.2 Examples of big-oh

- $2n^2 + 10n - 1 = O(n^2)$
- $e^n + n^{1000} = O(e^n)$
- $e^n + n^{1000} \neq O(n^{1000})$
- $O(1) < O(log_2 n) < O(n) < O(n^2) < \ldots < O(n^{1000}) < \ldots < O(e^n) < O(n!) < O(n^n)$

11.2.3 Some Standard Complexities

- A *polynomial-time algorithm* has complexity $O(n^m)$ for $m \geq 0$ an integer.
- A *linear-time algorithm* has complexity $O(n)$.
- A *constant-time algorithm* has complexity $O(1)$.
- A *logarithmic-time algorithm* has complexity $O(log_2 n)$.
- An *exponential-time algorithm* has complexity $O(a^n)$ for $a > 1$ (much harder than a polynomial-time algorithm).
- A *factorial-time algorithm* has complexity $O(n!)$.

11.3 Examples

11.3.1 Examples: Code segments

- For a sequence of statements, complexity is $O(1)$.
- For the following code segments, what is the time complexity with regard to n (assuming that all non-output statements are negligible)?
- Example 1:

```
int i, j;

for (i = 0; i < n; i ++)
  for (j = 0; j < n; j ++)
    cout << i*j << endl;
```

- Answer: We should find out how many times the inner output statement should be run.

  The outer `for` has n loops (i from 0 to n). For each loop we need to do another `for` in which we have n loops too (j from 0 to n). Thus there are $n \times n$ loops for two `for`s. Hence the final time complexity of this piece of program is $\mathbf{O(n^2)}$

- Example 2:

```
int i, j, k;

for (i = 0; i < n; i ++)
  for (j = 0; j < i; j ++)
    for (k = 0; k < j; k ++)
      cout << i*j*k << endl;
```

- Answer:

For the most inner `for` when both i and j are fixed we should do j output statements, i.e., k from 0 to $j-1$. In the middle `for` block, j is limited to 0 to $i-1$ where i is fixed. Thus to finish two inner `for` block we need to do

$$T(i) = 0 + 1 + 2 + \cdots + (i-1) = \frac{(i-1)i}{2}, \qquad \text{for a given } i$$

output statements. Now let us vary i from 0 to $n-1$. In order for us to finish outer `for` block, the total number of output statements to be implemented is

$$\begin{aligned}
T &= T(0) + T(1) + ... + T(n-1) \\
&= 0 + \frac{1(1-1)}{2} + \frac{2(2-1)}{2} + \frac{3(3-1)}{2} + \cdots + \frac{(n-1)(n-1-1)}{2} \\
&= \sum_{i=1}^{n-1} \frac{i(i-1)}{2} = \frac{1}{2}\sum_{i=1}^{n-1} i^2 - \frac{1}{2}\sum_{i=1}^{n-1} i \\
&= \frac{1}{12}(n-1)n(2n-1) - \frac{1}{4}n(n-1) \\
&= \frac{1}{6}n(n-1)(n-2) \\
&= O(n^3)
\end{aligned}$$

11.3.2 Space Complexity: Recursive fibonacci function

- A function call requires a constant k amount of stack space.
- For the recursive *fibonacci* function below, what is the **space complexity** with regard to n.

```
int fib(int n) {
  if (n <= 2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

- Interestingly enough, the answer is $\mathbf{O(n)}$
- Take $n = 5$ as an example. When the program begins to call `fib(5)`, in the body of `fib(5)` the program needs to do `fib(4)` first. The program suspends calculation and begin to call `fib(4)` before the addition (i.e., `fib(4)+fib(3)`). At this stage, the program will store the information in the current module, i.e., the parameter 5 will be stored in the function stack by the operating system. Please note that at this stage nothing happens to `fib(3)`.

  As the program is running on `fib(4)`, it is stepping into the second level of the recursive block. In the second level block the program is going to call `fib(3)` (Please note this `fib(3)` is not the `fib(3)` at the upper level needed for `fib(5)`. When the program moves into this `fib(3)` the system will store the information of its caller `fib(4)`, i.e., 4 in the stack.

  Now the program is running on `fib(3)`. In the same way, within the body of `fib(3)` the program is going to call `fib(2)` first. At this point, caller `fib(3)`'s information (i.e. 3) will be pushed on to the stack and then the program is about to process `fib(2)` call.

In processing `fib(2)` the program meets the terminal condition and return the value 1 as the returned value from the function call `fib(2)`. Then the program starts the second function call `fib(1)` in the body of `fib(3)`. Similarly the call `fib(1)` returns with a value of 1.

Now the stack uses 3 unit spaces to hold the information 5, 4 and 3.

Then the program returns to the caller `fib(3)`, thus the information 3 will be popped up from the stack and `fib(3)` returns the value of `fib(2)+fib(1)`. And the program comes back to the body of the caller `fib(4)`. In the body the program needs to process the second function call `fib(2)` in which the terminal condition is arrived. After calculating `fib(3)+fib(2)`, the program is going back to the upper level caller `fib(4)`.

Then the information 4 will be popped up from the stack and the program just finishes the first function call in body of calling `fib(5)` and then moves to the second function call `fib(3)`. Within `fib(3)` the program needs to push the information 3 onto the stack again in order to process the inner calls `fib(2)` and `fib(1)`. Now the stack holds the information 5 and 3.

After finishing the process of calls `fib(2)` and `fib(1)` the program pops out the information 3 and returns to the body of its caller `fib(5)` and calculates the addition `fib(4)+fib(3)`.

Finally the program pops up the information 5 and finishes the calling `fib(5)`.

In the whole procedure only 3 $(5-2)$ unit spaces are needed to implement recursive function call `fib(5)`. Generally in the procedure of computing `fib(n)` the program needs about $n-2$ unit spaces for the stack. Thus the space complexity of `fib(n)` is $O(n)$.

Chapter 12

Recursion and Iteration

In general, there are two approaches to writing repetitive algorithms. One uses iteration; the other uses recursion. Recursion is a repetitive process in which an algorithm calls itself. In this lecture, we study recursion and iteration as well as backtracking algorithms.

12.1 Recursive Functions

12.1.1 Recursive Tasks

- A *recursive task* is one that calls itself. With each invocation, the problem is reduced to a smaller task (*reducing case*) until the task arrives at some *stopping* or *base case*, which stops the process. The condition that must be true to achieve the terminal case is the *stopping condition*.
- A recursive function only works if each recursive call is to a simpler task.
- For a number of problems, recursive algorithms are simpler.
- See *Figure 9.1* on *page 420* and *Figure 9.2* on *pages 423*.

12.1.2 General algorithm for recursion

- Instead of a looping control structure, recursive functions use an `if` or `if ...else` statement, such as

```
if (stopping_condition)
    terminal_case
else
    reducing_case
```

or

```
if (!stopping_condition)
    reducing_case
```

- Example
  - The first example is the factorial function: $n!$.
  - This is defined as:

$$n! = \begin{cases} n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 & \text{for } n > 0 \\ 1 & \text{for } n = 0 \end{cases} \tag{12.1}$$

  - This gives us the recursive form:

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n(n-1)! & \text{for } n > 0 \end{cases} \tag{12.2}$$

- Recursive version of $n!$

```
double recFact(int n) {
  if (n == 0)
    return 1;

  return n * recFact(n-1);
}
```

- Iterative version of $n!$

```
double itFact(int n) {
  double retVal = 1.0;
  int i;

  for (i = 1; i <= n; i ++)
    retVal *= i;
}
```

12.1.3 Tracing Recursive Calls

- The computer keeps track of function calls in the following way: When a function call is encountered, the first step is to save some information that will allow the computation to return to the correct location after the function call is completed.
- The computer also provides memory for the called function's formal parameters and any local variables that the called function uses.
- Next, the actual arguments are plugged in for the formal parameters, and the code of the called function begins to execute.
- If the execution should encounter another function call — recursive — then the first function's computation is stopped temporarily. This is because the second function call must be executed before the first function call can continue.
- Information is saved that indicates precisely where the first function call should resume when the second call is completed.

- The second function is given memory for its own parameters and local variables. The execution then proceeds to the second function call.
- When the second function call is completed, the execution returns to the correct location within the first function. And the first function resumes its computation.
- Suppose that we want to calculate the factorial of 5.
- We call the function `recFact(5)`, then the computer allocates memory for the formal parameter `n` (now no local variables at all) and copies the actual argument value 5 into memory of the formal parameter `n`.
- As the value of `n` is nonzero, then the function executes the second statement in its body, i.e., the `return` statement.
- There it encounters another function call `recFact(4)`. Then the computer stops the computation of the first function call and begins processing the second function call. At the same time, information is saved that indicates the point of the first function computation.
- The computer allocates new memory for the formal parameter `n` and plugs value 4 into it. Note memory for `n` here is different from memory for `n` in the first function call which holds a value of 5.
- Then the computer begins processing the second call in which the third function call is encountered. Then the same procedure goes through. Now suppose the second function call finishes its job, then the second function call returns its result to the body of the first function call.
- Finally the first function call resume its computation from the stopping point, i.e., computing 5 times the result from the second function call, and then returns the product. Now the first function call finishes its job.

12.2 Examples

12.2.1 The Problem

- In this example we consider a recursive function for deleting an object from a black and white image.
- Two black pixels are in the same object if we can get from one to the other with horizontal and vertical moves without going over any white pixels.
- Thus the image in Figure 12.1 has two objects.

12.2.2 Algorithm: Deleting an object from the image

- To delete an object we specify a pixel in `(row, col)` format.
- The base case occurs if the pixel is outside the bounds of the image, or if it is already white.

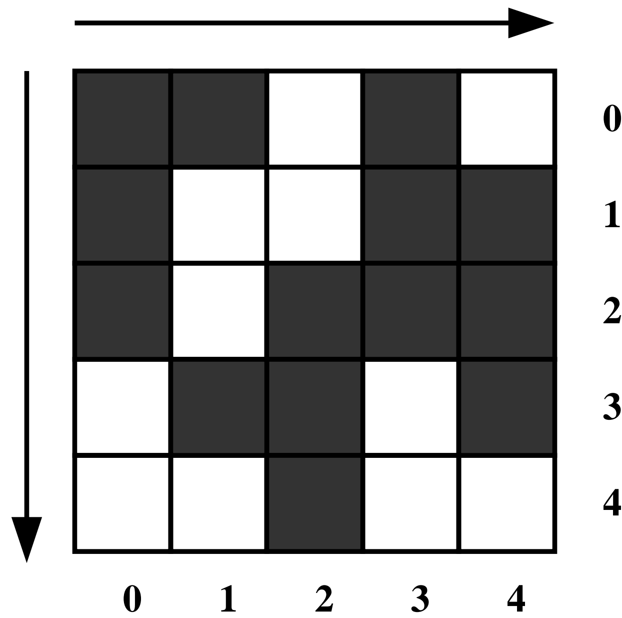


Figure 12.1: Initial image

- If not then we white out the pixel and then call the function recursively for the pixels directly above, below, to the left and to the right of the current pixel.
- Two base cases:
  - Pixel is outside the image.
  - Pixel is already white.
- Recursive case: make the pixel white and call function for each surrounding pixel.
- eg. Figure 12.2 shows the result when we delete the object at pixel (3,2).

12.2.3 Defining function ErasePic

- Here is a recursive algorithm for the `ErasePic` function.

```
ErasePic algorithm for pic at (row, col)
if (row, col) is inside pic and
   (row, col) is black
  change (row, col) to white
  call ErasePic for:
    (row - 1, col    ),
    (row + 1, col    ),
    (row    , col - 1) and
    (row    , col + 1)
```

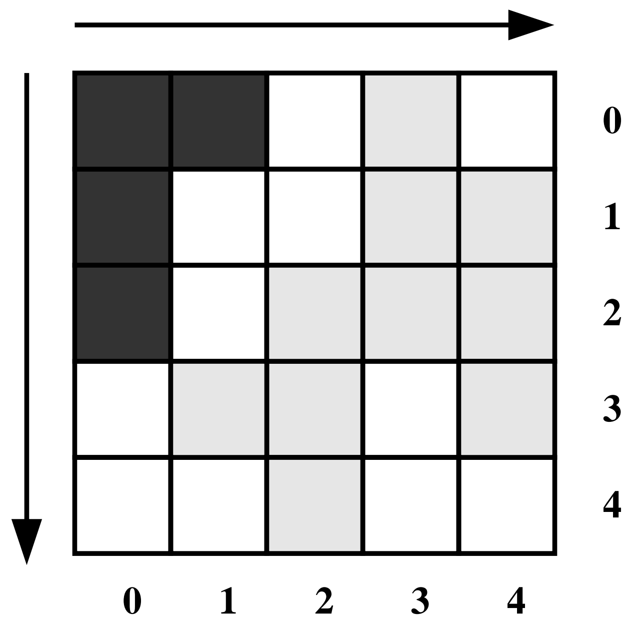


Figure 12.2: Final image

12.2.4 Transformed into C++

- We use a 2D array of type `int` to represent an image. If the pixel is white, then the corresponding element in the array is 0; if the pixel is black, the element has 1.
- The code:

```
const int SIZE  = 5;
typedef int pic_t[SIZE][SIZE];

void ErasePic(pic_t pic, int row, int col) {
   if ((0 <= row) && (row < SIZE) &&
       (0 <= col) && (col < SIZE) &&
       (pic[row][col] == 1)) {
      pic[row][col] = 0;
      ErasePic(pic, row + 1, col    );
      ErasePic(pic, row,     col + 1);
      ErasePic(pic, row - 1, col    );
      ErasePic(pic, row,     col - 1);
   }
}
```

- Suppose we call the above function at pixel (3,2), then can you trace the order of pixels changing from black to white?

12.3 Recursion vs. Iteration

- In Favour of Iteration:
  - Recursive functions use more stack space than their iterative counterparts.
  - Recursive functions can require more CPU time.
- In Favour of Recursion:
  - The most natural and simple solution to a problem may be recursive.
  - Some compilers can automatically convert some recursive functions to equivalent iterative functions.

12.4 Backtracking Algorithms

12.4.1 Principle

- A *backtracking algorithm* is an algorithm where we guess a solution, and back up if an impasse is reached.
- Like navigating a maze
  - Try one direction.
  - Back up if we reach a dead end.
- *Backtracking* should be used when:
  - We have more than one choice.
  - We don't know which choice will work.

12.4.2 Eight Queens Problem

- A *chessboard* is an 8x8 board.
- A *queen* can move along its row, column and it diagonals.
- The problem is to place eight queens on a chessboard such that no queen can take another.
- We can do this using backtracking.

12.4.3 Different Strategies

- Consider all possible combinations:

  $\binom{64}{8} = 4,426,165,368$ combinations.
- Restrict attention to *one queen per row*:

  $8^8 = 16,777,216$ combinations.
- Restrict attention to *one queen per row & column*

$8! = 40,320$ combinations.

- We can also immediately discard all guarded diagonals.

12.4.4 Backtracking Algorithm for Problem

- This algorithm is recursive, so it still has the same basic form given earlier. However, it may try several recursive cases and then a base case.
- We consider the board column by column.
- For each column we find an unguarded position for the queen.
- We repeat the process for the next column (*recursive case*).
- If this fails then we find another unguarded position for this column.
- If there are no more unguarded positions then we signal failure and exit (*base case*).
- If there are no more columns to consider then we signal success and exit (*base case*).

12.4.5 General Algorithm for Backtracking

```
if (stopping_condition)
  stopping_case
else
  while (more_guesses)
    try_guess_reduce
```

or

```
if (!stopping_condition)
  while (more_guesses)
    try_guess_reduce
```

12.4.6 Algorithm for `addQueen`

```
addQueen algorithm for board at (row, col)

//check base cases
if col >= BOARD_SIZE then
   return with success  //we have finished all columns
if row >= BOARD_SIZE then
   return with failure  //we have checked the whole column col
//base cases end

//check if (row, col) is guarded
if isGuarded(board, row, col) then
   return the result of addQueen for board at (row+1,col)
```

```
add a Queen to board at (row, col)

//check agian to see if the puzzle is solved
if success is returned from addQueen for board at (0,col+1)
   then return with success

//if all places on col+1 are under attack, we should
//backtrack to the last step
remove the Queen at (row, col) from the board

//try the next row on column col
return the result of addQueen for board at (row+1, col)
```

The job of `addQueen` function is to play the game at the location `(row, col)`and report the result (`success` or `failure`) from the location. Within the function, we first check if the location is outside the board. If so, then the game is over and the function returns the game status (`success` or `failure`). Then check if the location is guarded by the placed queens. If guarded, then we play the game at the next location `(row+1, col)`; if not, we put a new queen at the location. After that we further play the game from a new column, that is, from the new location `(0, col+1)`. If the returned result is failure from that location, that means the location `(row, col)` is not a good location for the new queen. Hence we go back one step by removing the queen at `(row, col)` and try the next location on the same column, that is, the location `(row+1, col)`.

Chapter 13

Algorithms for Search

In this lecture, we talk about the concept of *key* and comparing algorithm, Sequential search and Binary search (two versions) and their properties and conclusions.

13.1 Why Search?

Searching a list of values is a common computational task. An application program might retrieve a student record, bank account record, credit record, or any other type of record using a search algorithm.

13.1.1 Objectives

- Search an array for a target using two algorithms:
  - Sequential search.
  - Binary search.
- Compare efficiency of algorithms.
- We are not limiting ourselves to one specific type of array.

13.1.2 Keys

- To search an array, we must be able to compare elements.
- This is done via a *key*. A key is something which can be used to identify or clarify objects/elements. It's unique for objects of classes.
- For simple types like `int`, `char`, and `char*`, the key may be the element itself.
- For more complex types such as structures, this *key* may be one specific field.
- For example, in the following structure, we could use `studentNumber` as the key.

```
struct student {
  char firstName[32];
  char lastName[32];
  int  studentNumber;
};
```

13.1.3 General Principles

- The searching algorithms should return:
  - The *index* of the target in the array if it is found.
  - -1 if the target is not found.
- The algorithms are compared by the number of *key* comparisons they require.

13.2 Sequential Search

13.2.1 Principle

- Also called *Linear search*.
- Begin at the start of the array.
- Search the array, element by element.
- Finish when:
  - The target is found.
  - The end of the array is reached.
- The technique is often used because it is easy to write the algorithm code and is applicable to many situations including both sorted and unsorted arrays.

13.2.2 Algorithm for Sequential Search of an unsorted array

We search for a desired item x in an array a with n array elements. The pseudocode for the algorithm is

```
i = 0
found = FALSE
while ( (i < n) and (not found) ) do the following
{
    if (x is a[i])
       found = TRUE
    else
       i = i + 1
}
```

```
if (i == n)
  i = - 1
return  i
```

13.2.3 Algorithm Analysis: Number of Comparisons required

- How many comparisons are required for a sequential search?
- The perfect answer depends on the distribution of elements in the array. Usually when we analyze the performance of an algorithm, we consider the "hardest" case. This is called the *worst-case*.
- For the sequential search algorithm, the worst-case search occurs when the target is the last element in the array. In this case, the algorithm compares every element in the array with the target thus the algorithm takes n comparisons.
- Another worst-case is that the target is not in the array at all. In this case, the algorithm has to complete comparison with each element in the array.
- The best case occurs when the target is the first element in the array, thus only one comparison operation is needed.
- On average for an array of length n:
  - A successful search: $\frac{n+1}{2}$ (the best case: 1, and the worst case: n).
  - An unsuccessful search: n (no such element x in the array a)

13.2.4 Example: Sequential Search on Character Array

- For a character array, the *key* is just the element.
- We search an array of size $n = 8$ for the element 'M'.

a:	**'R'**	**'A'**	**'F'**	**'P'**	**'M'**	**'B'**	**'X'**	**'V'**
index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

- In this example, the algorithm needs five comparisons between the target and the elements in the array.
- If the target x is 'V', the successful search needs 8 comparison.
- If the target x is 'W', then the algorithm needs 8 comparisons but unsuccessful.

13.3 Binary Search

Sequential search is easy to implement, easy to analyze, and fine to use if we are only searching a small array. However, if a search algorithm will be used over and over, it is worthwhile to find a faster algorithm. The algorithm, called *binary search*, may be used.

13.3.1 Problem and Idea

- Cuts problem in half with each iteration.
- Array must be sorted for the algorithm to work.
- We consider the middle element. Three cases:
  - Larger than target:

    Search the bottom half of the array.
  - Smaller than target:

    Search the top half of the array.
  - Equal to target:

    Found it!

13.3.2 Algorithm for Binary Search of a sorted array

We search for a desired item x in an array a with n array elements. The pseudocode for the binary search algorithm is

```
first = 0
last = n-1
found = FALSE
while ( (first <= last) and (not found) ) do the following
{
    mid = (first + last)/2
    if ( x < a[mid] )
         last = mid - 1
    else if ( x > a[mid] )
         first = mid+1
    else
         found = TRUE
}
if ( not found ) then
    mid = -1
return  mid
```

13.3.3 Example: Binary Search on Integer Array

- For an integer array, the *key* is just the element.
- We search an array of size 9 for the element 28.

a:	3	7	8	9	14	22	24	28	31
index:	0	1	2	3	4	5	6	7	8

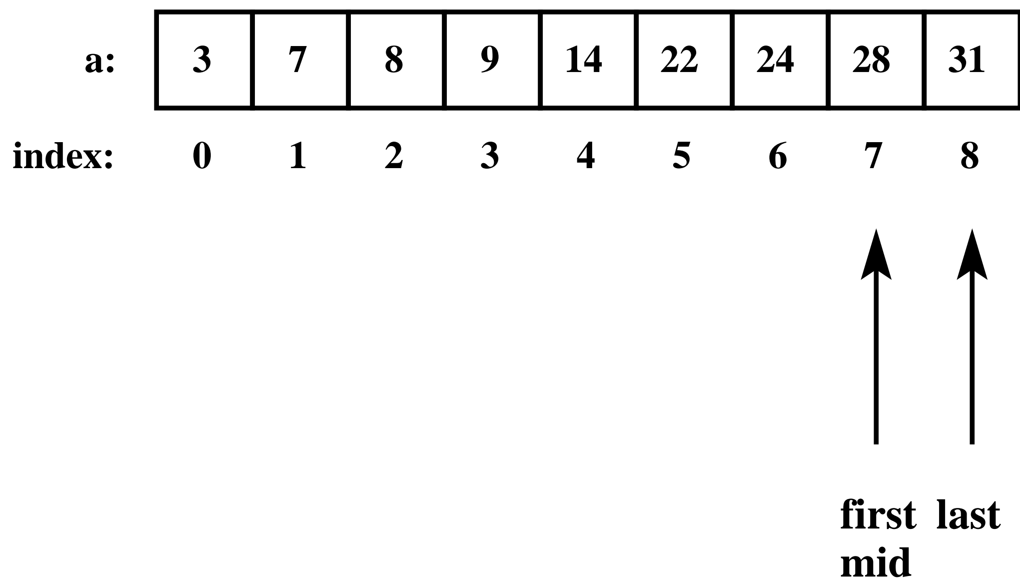
first (index 0), mid (index 4), last (index 8)

- Step Two

a:	3	7	8	9	14	22	24	28	31
index:	0	1	2	3	4	5	6	7	8

first (index 5), mid (index 6), last (index 8)

- Step Three



13.4 Alternative Binary Search

13.4.1 Idea

- In this algorithm, we only do one comparison per iteration.
- We consider the middle element. Two cases:
  - Less than target or equal to target:

    Search the top half of the array.
  - Greater than target:

    Search the bottom half of the array (including middle element).

13.4.2 Alternative Algorithm for Binary Search

- We search an array a for the target x:

```
first = 0
last = n-1
while ( first < last ) do the following
{
     mid = (first + last) / 2
     if ( x > a[mid] )
         first = mid+1          else
         last = mid
}
if (first = last and x = a[last])
```

```
        exit with last
    exit with -1
```

- Question: What will happen if the key is the middle element at first comparison?
- Consider two special cases: (1) the key is the first element and (2) the key is the last element.

13.5 Pros and Cons of Searching Techniques

- Sequential Search:
  - Works on both sorted and unsorted arrays.
  - Requires a lot of comparisons.
- Binary Search:
  - Is much faster than Sequential Search.
  - Only works on sorted arrays.
- Number of Comparisons (Successful Search)

Size	Sequential	Binary 1	Binary 2
10	5.5	5	3
100	50.5	9	7
1000	500.5	14	10
10000	5000.5	19	13
100000	50000.5	24	17

- Number of Comparisons (Unsuccessful Search)

Size	Sequential	Binary 1	Binary 2
10	10	6	3
100	100	10	7
1000	1000	15	10
10000	10000	20	13
100000	100000	25	17

Chapter 14

Hashing and Complexity

14.1 Hashing

14.1.1 Definitions

- A *table* is a finite sequence of records such that each entry has a *key* which uniquely identifies it.
- There are three basic operations for a table:
  1. Insert an entry into table
  2. Delete an entry from table
  3. Search for key on the table
- The *load factor*, is a measure of the saturation of a table. It is the ratio of the number of items in the table and the number of table locations.

$$\text{load factor} = \frac{\text{number of items in table}}{\text{number of table locations}}$$

- The table structure is the main component of a relational database. A relational database is composed of many tables, called relations.
- Example Table

Name	Year	GPA
Adams, Keith	3	3.21
Blackwell, Henry	1	2.00
Davis, Susan	1	3.50
Hinkel, Chris	2	1.94
Jordan, Ann	4	3.05
Patterson, Lynn	1	2.00
Suarez, Rosa	3	2.82
Williams, George	2	3.64

14.1.2 Different Implementations of Tables

There are a number of implementations of ADT table. A dynamic linked list or an array of records — in both cases with ordered or unordered data — or a binary search tree can provide the implementation of table. That is

1. Array:
   (a) Sorted
   (b) Unsorted
2. Linked List:
   (a) Sorted
   (b) Unsorted
3. Binary Search Tree
4. Hash Table (Covered in this Lecture Note)

The characteristics of a particular application, including the frequency of doing each operation, determine the best implementation of ADT table.

To match a table implementation with an application, we must have a clear understanding of each implementation's time and space complexities.

14.2 Hash Tables

14.2.1 Definition

- A *hash table* is a table that uses a function **h**, called a *hash function*, to compute a record's numerical location in the table from its key.
- In the absence of collisions:
  - *Insertion:* Insert the entry with **key** at the location h(**key**)

- *Deletions:* Delete the entry with **key key** at the location h(**key**)
- *Retrieve:* Check if the entry with **key** is at the locationh(**key**)

14.2.2 The hash function

- We use the *hash function* to determine the location for a record or jump immediately to the desired record.
- This *direct access* approach provides very efficient insertions, deletions and retrievals.
- In the *remainder method* of hashing, the hash function has the form.

  $$hash(k) = k\%c$$

  where k is a key, and c is a positive constant.
- Typical choice for c is the table size.

  $$hash(key) = key\%TABLE_SIZE$$

- Note: The hash function may map two different keys to the same index. This leads to the possibility of collisions.

14.2.3 Example: Hash table of records

hash(k) = k % 11

Index	Key	Information
0	3542	. . . .
1		
2	5370	. . . .
3	7406	. . . .
4		
5	9179	. . . .
6		
7		
8		
9	6598	. . . .
10	5081	. . . .

14.2.4 Good Table Size

- In general, primes give the best table sizes. Suppose we pick a non-prime for our table size, eg: 14. The factors of 14 are 1, 2, 7, 14. Choose the hash function: $hash(k) = k\%14$. Then even number keys will be hashed to even indices and odd number keys hashed to odd indices.
- We wish numbers to be dispersed in a semi-random fashion throughout the table. Eg: Suppose all keys were even, then we would only use half of the table.

14.3 Collision Problems

14.3.1 Resolving Collisions

- In a hash table, a *collision* occurs when two keys hash to the same location
- A *probe* into a hash table is a check of a location for a key. Many collision-resolving methods are possible:
  - Linear probing
  - Quadratic probing
  - Random probing
  - Chained addressing

14.3.2 Open Addressing

- *Open addressing* is a method of finding an open location for insertion into a hash table after a collision has occurred.
- *Linear probing* is an open addressing technique. If there is a collision, we continue from the hash location on, looking for the next available position.

  h, h+1, h+2, ...
- When using open addressing, each location is assigned one of three states:
  - **Occupied:** Currently in use
  - **Empty:** Has not been used
  - **Chain:** Once occupied, but available now

14.3.3 Linear Probing: Insertions

Insert element with key 3093

hash(k) = k % 11 3093 % 11 = 2

Index	Key	Information
0	3542	. . . .
1		
2	5370	. . . .
3	7406	. . . .
4	3093	. . . .
5	9179	. . . .
6		
7		
8		
9	6598	. . . .
10	5081	. . . .

14.3.4 Linear Probing: Deletions

Delete key 3645

hash(k) = k % 11 3645 % 11 = 4

Index	Key	Status	Information
0	4719	OCC.	. . . .
1		EMPTY	
2		EMPTY	
3		EMPTY	
4	3194	OCC.	. . . .
7	3645	CHAIN	
6	4747	OCC.	. . . .
5	2072	OCC.	. . . .
8		EMPTY	
9	4200	OCC.	. . . .
10		EMPTY	

14.3.5 Linear Probing: Searching

Search for key 2072

hash(k) = k % 11   2072 % 11 = 4

Index	Key	Information
0	4719	. . . .
1		
2		
3		
4	3194	. . . .
5	3645	. . . .
6	4747	. . . .
7	2072	. . . .
8		
9	4200	. . . .
10		

14.3.6 Clustering

- Any key that maps to a location h follows the same path, searching for an available cell.
- If many keys map to the same location, we will get clustering.
- The clustering of records in a hash table that follow the same path from a hash location is called *primary clustering*
- When a probing path for one key merges with other hash values and paths, *secondary clustering* occurs.

14.3.7 Quadratic Probing

- The probe sequence examines the locations that are the squares of the integers beyond the original hash position h, modulo $TABLE_SIZE$.

  h + 1, h + 4, h + 9, h + 16, ...

- Next location depends on number of probes so far.
- This method reduces secondary clustering, but does not reduce primary clustering, and it may not examine every location

14.3.8 Quadratic Probing Example: Insertions

Insert key 3035
hash(k) = k % 11   3035 % 11 = 1

Index	Key	Information
0		
1	3312	. . . .
2	2191	. . . .
3		
4		
5	2447	. . . .
6		
7		
8		
9	3606	. . . .
10	3035	. . . .

14.3.9 Random Probing

- Random number generator computes next location to be examined. The key is used to seed the random number generator so that the probe sequence can be duplicated during searches.
- This probing method reduces both primary and secondary clustering. It is usually slower because of time taken in generating random numbers.

14.3.10 Load Factor

With any open addressing method of collision resolution,

- As the table fills, there can be a severe degradation in the table performance.
- Load factors between 60% and 70% are common. Load factors $>$ 70% are undesirable.
- The search time is dependent only on the load factor, *not* on the table size. We can use the desired load factor to determine appropriate table size:

$$\text{table size} \quad = \quad \text{smallest prime} \geqslant \frac{\text{number of items in table}}{\text{desired load factor}}$$

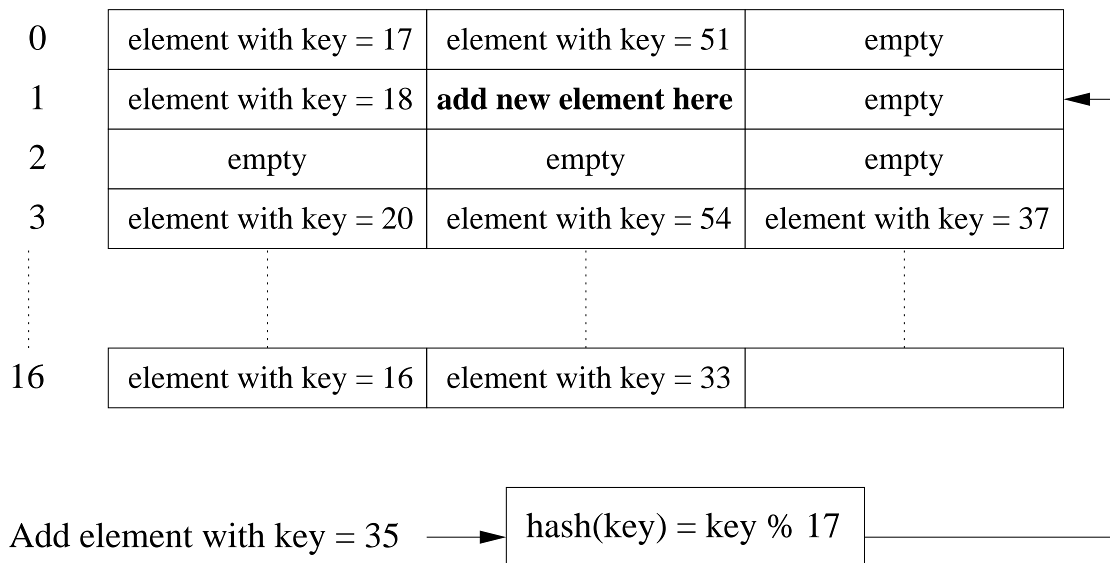
14.4 Bucket Technique and Chained Addressing

14.4.1 An Alternative to Open Addressing Techniques

- Definition:

  A *bucket* is a collection of elements associated with a particular hash location.

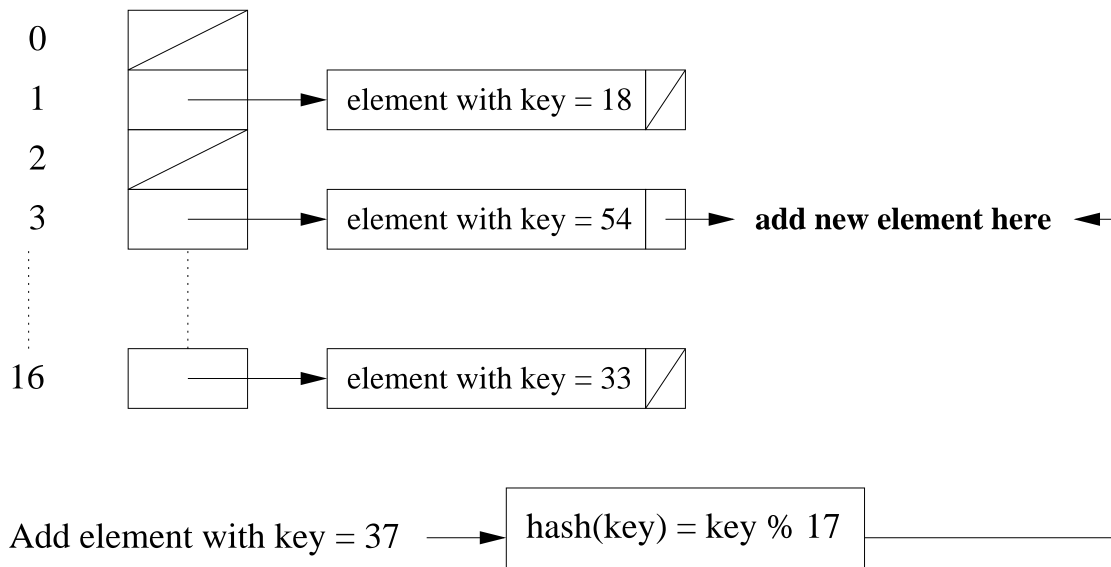
- Allow collisions to produce many entries at the same location. When the bucket becomes full, we must again deal with handling collisions.
- Example: Hashing with Buckets



14.4.2 Chained Addressing

Another technique to resolve collision is chained addressing which uses dynamic linked lists. Moreover, the implementation expands the table as needed and eliminates secondary clustering.

- Each table position is a linked list.
- A *chain* is a linked list of elements that share the same hash location.
- Collisions are resolved by adding the element to the linked list at that location.
- Chained Example


0
1
2
3
16
element with key = 18
element with key = 54
add new element here
element with key = 33
Add element with key = 37
hash(key) = key % 17

Chapter 15

Stacks and Their Application

In this lecture we talk about the stack structure. Stack is widely used in many computer techniques. For example recursion is implemented in the computer by use of a run-time stack. The stack is appropriate in situations when the last data placed into the structure should be the first processed.

15.1 Stack Structure

15.1.1 What is a Stack?

- A *stack* is a data structure in which the last data item placed into the structure is the first accessed.
- All activity occurs at one end – the *top* – of the stack.
- The operation to place a data item on the top of the stack is *push*.
- The **pop** operation removes a data item from the stack.
- **LIFO** is an acronym for *L*ast *I*n *F*irst *O*ut.
- The *top* operation returns the top of the stack without removing it
- The **IsEmpty** operation returns *true* if the stack is empty (i.e. has no elements).
- Think of a stack of plates at a salad bar:
  - We can add to the top of the stack (push)
  - We can remove the top plate (pop)
  - To access the nth plate, we must first remove the n-1 plates above it.

15.1.2 The STL Stack Class

- Because different stacks may store different types, a Class Template for a stack is desirable. We need one template parameter: `class Item` (the type of data to be stored in the stack)

- The C++ Standard Template Library (STL) has a stack class. You can simply use it to define your own stack object. For example, the following statement declares a stack for `string`

```
stack<string> myStrStack;
```

- The main methods for the standard stack are

```
bool empty();  //check if the stack is empty
void push(Item& entry);   //add a data into the stack
Item Pop();   //take the object/data from the stack
Item& top();  //index to the top element
```

15.1.3 Our Version for Stack Template

- Array implementation of a stack

```
template <class Item>
class stack {
   public:
       static const size_type CAPACITY = 30;
       stack() { used = 0;  }
       void push(const Item& entry);
       void pop();
       bool empty() const
       { return (used==0); }
       Item top() const;
   private:
       Item data[CAPACITY];
       size_type used;
};
```

- Dynamic Implementation of a stack

```
template <class Item>
class stack {
   public:
       conststatic const size_type CAPACITY = 30;
       stack() { top_ptr = NULL; }
       stack(const stack& source);
       ~stack(){ list_clear(top_ptr); }
       void push(const Item& entry);
```

```
        void pop();
        void operator=(const stack& source);
        bool empty() const
        { return (top_ptr==NULL); }
        Item top() const;
    private:
        node<Item>* top_ptr;
};
```

15.2 Applications

15.2.1 Where are Stacks used?

- Stack frames are used to store return addresses, parameters, and local variables in a function calling
- Computer graphics (OpenGL). The sequence of transformations uses a *last-specified, first-applied* rule. Thus, a stack of transformations is maintained.
- Robotics: Instructions are stores in a stack. We can apply stack controllers such as repeat loops to these stacks.
- Architecture of computers uses stacks to do arithmetics (eg. Intel FPU).

15.2.2 An example: Reversing a string

Two possible solutions:

- Use recursion
- Use Stack

```
while more input do
  read character
  push it onto stack
end

while not stack is empty do
  pop character
  write it to the screen
end
```

15.2.3 From Infix to Postfix

- Infix notation: $(A + B) * C$
- Postfix notation: $AB + C*$

- We can use a stack to change notations.
- Algorithm

```
while not end-of-expression do:
  read next symbol
  cases for symbol:
    an operand: write it
    '('       : push '('
    ')'       : pop & write operators until
                '(', then pop '('
    '*','/'   : pop & write operators until
                '(', '+' or '-'
                then push '*' or '/'
    '+','-'   : pop & write operators until
                '(', then push '+' or '-'
    EOE       : pop & write all operators
```

15.2.4 Evaluation of Postfix

- When an expression is in postfix notation, the computer pushes the operands onto the stack as it reads the expression from left to right. When an operator is encountered, the top two items are popped from the stack, the binary operation evaluated, and the result pushed back onto the stack.
- This way of evaluation is used in the Intel FPU.
- Algorithm

```
while not end-of-expression do
  read next symbol
  if symbol is an operand
    push it onto stack
  else (symbol is an operator)
    pop two operands,
    apply operator to them
    push the result
  endif

pop the result
```

Chapter 16

Queues and Their Application

In this lecture, we describe the queue structure which, for example, models a line of people waiting at the salad bar, discuss the abstraction data type definition of queue as well as the appropriate operation on the queue structure.

16.1 Queues

16.1.1 Definition of a Queue

- A list where additions and deletions are done at opposite ends.
  - `enqueue/push`, add object to the *rear* of the queue
  - `dequeue/pop`, remove object from the *front* of the queue
- *FIFO*: *F*irst *I*n *F*irst *O*ut. There are two implementations:
  - Arrays: covered later
  - Linked Lists: covered later
- Additional operations:
  - `empty`: Contains no elements
  - `full`: Can not add more elements
  - `front`: returns object at **front** but doesn't delete it
- Analogy: consider a line of people waiting to be served:
  - New arrivals join the rear of the line.
  - People leave the line at the front to be served
  - People are served in the order that they arrive (FIFO)

16.1.2 The STL Queue Class Template

- The C++ Standard Template Library (STL) has a queue class. You can simply use it to define your own stack object. For example, the following statement declares a stack for `string`

```
queue<string> myStrQueue;
```

- The main methods for the standard stack are

```
bool empty();  //check if the queue is empty
void push(Item& entry);   //add an item into the queue
Item pop();   //remove the object/data from the front
Item& front();  //returns the from item
```

16.1.3 Array Implementation of Queue

- We use `first` and `last` to indicate the indices of the front and rear elements, respectively.
- The template class is defined as

```
template <class Item>
class queue {
   public:
     //queue methods
   private:
     Item data[CAPACITY];
     size_type first;
     size_type last;
     size_type count;
};
```

where `count` is the number of items in the queue.

- When adding elements to the queue, the index `last` increases. When deleting elements from the queue, the index `first` also increases. That is, each time we enqueue an element, the queue grows to the right (ie. larger indices). Each time an element is dequeued, the queue shrinks from the left. Thus, the entire queue will slowly *drift* to the right. Eventually, we will reach the end of the array, and no longer be able to add elements to the queue even there are space on the left hand side of the array.
- To solve this problem, consider a queue of people waiting to be served. Each time someone leaves the queue from the front, everyone else shuffles forward to fill the gap. Similarly, for an array based queue, we could shift each element of the queue one index to the left each time we dequeued something.

- A more efficient solution is to use a circular array.
- The array wraps around, so that if we reach the end of the array we start enqueueing elements at the start. Of course, if the queue is full, we still won't be able to enqueue any more elements.

16.2 Queue Applications

16.2.1 Computer Science

- Simulations
  - People waiting to be served
  - Airport Traffic: Planes waiting to take off & land.
- Operating System
  - Jobs wait in queue to execute
    - First Come First Served: **FCFS**
    - Round Robin scheduling

16.2.2 Recognizing palindromes

- Recall a stack is a Last In First Out structure.
- To recognize palindromes, we first enter the line into both a stack and a queue of characters. Then we compare the output of both structures char by char. Because a stack is a **LIFO** structure and a queue is **FIFO**, the order of the stack output is reversed, while the queue output is in order.
- Algorithm: Palindrome

```
while more input do:
  read character of input
  Push onto Stack
  EnQueue onto Queue
end while

while NOT stack is empty do:
  Pop char1 off Stack
  DeQueue char2 off Queue
  if (char1 <> char2) then
    return FALSE
  end if
end while

return TRUE
```

16.2.3 Evaluation of Infix

- Recall from lecture 15:
  - *Infix* & *Postfix* expressions
  - Algorithm: Infix → Postfix
  - Algorithm: Evaluation of Postfix
- We can combine these algorithms
- Algorithm: Evaluation of Infix
  - Convert *expr* to *Postfix*
  - Store in **Queue**
  - Evaluate stored *expr* & output

16.2.4 Simulation of customer service: (*pp. 381-396*)

- Consider a queue of car to wash
- Must specify certain constants:
  - *WashTime*: for one customer
  - *SimulationTime*: running time
  - *ProbabilityArrive*: rate of Arrival (how to decide if a new customer arrives)

    At a certain time, to determine if a new customer arrives, we obtain a random number between 0 and 1. If this number is less than `ProbabilityArrive`, we process an arrival by enqueuing a customer.
- Simulation is time based (discrete)
- A queue of customers is maintained. (Service is nothing but only the time decrement in this example)
- We can work out the average delay.
- Implementation: *pp. 390 - 396*
- Algorithm:

```
read WashTime, SimulationTime &
  ProbabilityArrive
initialize the clock & queue
WindowTimeLeft = 0

while (clock < SimulationTime) do:
  process possible arrival
  if (WindowTimeLeft = 0 AND
```

```
      queue is not empty) then
    serve car at front of queue
    WindowTimeLeft = TransactionTime
  end if
  if (WindowTimeLeft > 0) then
    decrement WindowTimeLeft
  end if
  increment clock
end while
```

Chapter 17

Dynamic Queues

17.1 Linked List Implementation of a Queue

17.1.1 Queue Template Class

- A queue can also be implemented as a linked list. One end of the linked list is the front, and the other end is the rear of the queue. Dynamic queues should be used if we can't predict the maximum size.
- Our approach uses two pointers: One points to the first node (`front_ptr`), and the other points to the last node (`rear_ptr`).
- Here is a class definition for the `queue` template class,

```
template <class Item>
class queue {
  public:
      queue(); //constructor
      queue(const queue<Item>& source);
                //copy constructor
      ~queue();  //destructor
      void pop();
      void push(const Item& entry);
      void operator=(const queue<Item>& source);
      bool empty() const { return (count==0); }
      Item front() const;
      size_type size() const { return count; }
  private:
      node<Item>* front_ptr;
      node<Item>* rear_ptr;
      size_type count; //total number of items
```

```
};
```

17.1.2 Implementation

- The constructor will initialize an empty queue. The implementation is simple

```
template <class Item>
queue<Item>::queue() {
    front_ptr = NULL;
    rear_ptr = NULL;
    count = 0;
}
```

- The copy constructor will create a new queue object from the given queue. The implementation is to use the `list_copy` function of the node class to copy items from the source to the newly created queue.

```
template <class Item>
queue<Item>::queue(const queue<Item>& source) {
    count = source.count;
    list_copy(source.front_ptr, front_ptr, rear_ptr);
}
```

- Before you read the text about the detail of destructor, can you give the implementation of the destructor `~queue()`
- The `push` function of the `queue` class adds a node at the rear of the queue. In the implementation we need to create the node for the given entry, to add the node just after the current rear node (via the insertion function of the node), and to move the `rear_ptr` pointing to the newly added node.

```
template <class Item>
void queue<Item>::push(const Item& entry) {
    if (empty()) {
        list_head_insert(front_ptr, entry);
        rear_ptr = front_ptr;
    } else {
        list_insert(rear_ptr, entry);
        rear_ptr = rear_ptr->link();
```

```
    }
    ++count;
}
```

- The pop function of the queue class removes a node from the front of the queue. After removing the second node will be the new front node. In the implementation we need to return the space for the first node to the system and to move the front_ptr to the second node.

```
template <class Item>
void queue<Item>::pop( ) {
    assert(!empty());
    list_head_remove(front_ptr);
    --count;
}
```

- The assignment operation is also important. Here is the implementation of the overloaded = operator

```
template <class Item>
void queue<Item>::operator=(const queue<Item>& source) {
    if (this == & source)
        return;
    list_clear(front_ptr);
    list_copy(source.front_ptr, front_ptr, rear_ptr);
    count = source.count;
}
```

- Finally we need a function to return the data information in the front node.

```
template <class Item>
Item queue<Item>::front( ) const
{
    assert(!empty());
    return front_ptr->data();
}
```

17.2 Priority Queues

17.2.1 Priority Queue

- Using a queue ensures that customers are served in the exact order in which they arrive. However, we often want to assign priorities to customers and serve the higher priority customers before those of lower priority.

- For example, a computer operating system that keeps a queue of programs waiting to use some resource, such as a printer, may give interactive programs a higher priority than batch processing programs that will not be picked up by the user.

- A *priority queue* is a container class that allows entries to be retrieved according to some specified priority levels. The highest priority entry is removed first.

- If there are several entries with equally high priorities, then the priority queue's implementation determines which will come out first.

17.2.2 Specifying Priority

- The C++ STL has a `priority_queue<Item>` template class for `Item` type data.

- If the value of the `Item` type can be compared with a built-in "less than" operator, then that operator can be used to specify a priority for each item in the queue.

- For your own class type `Item`, you may have defined your own "less than" operator (overloaded operator `<`)

- A programmer may define his own priority function other than the operator `<`.

- In the above meaning, a priority queue can be considered as a sorted queue.

17.2.3 Implementation Ideas

- Actually, a priority queue is a sorted queue in the decreasing order based on the order of the items. The highest priority item is placed at the front of the queue.

- A common priority queue implementation uses a tree data structure that achieves high efficiency.

- There are several less efficient alternatives. One possibility is to implement the priority queue as an ordinary linked list, where the items are kept in order from highest to lowest priority

Chapter 18

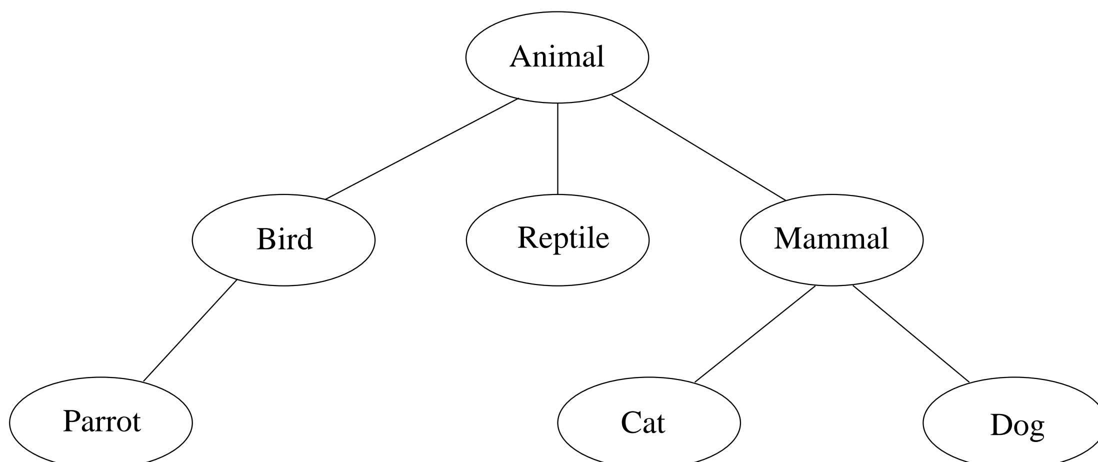
Trees and Binary Trees

Unlike lists etc., tree is a nonlinear structure. You have knowledge about tree from the course *discrete mathematics*. In this lecture we will discuss how to make abstract data type tree and its implementation methods.

18.1 The Tree Structures

18.1.1 What is a tree? Rooted tree?

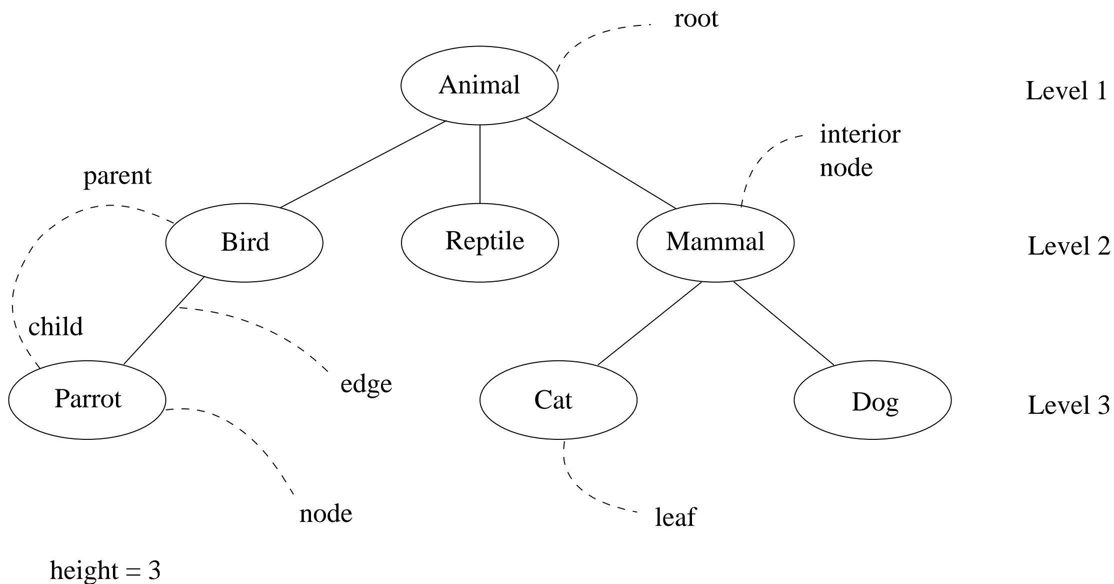
- Tree is a data structure that is hierarchical in nature
- For any nonempty tree, there is a hierarchical arangement with one node at the top. This representation is a *rooted tree*, the node at the top is called the *root*.



18.1.2 Tree definitions

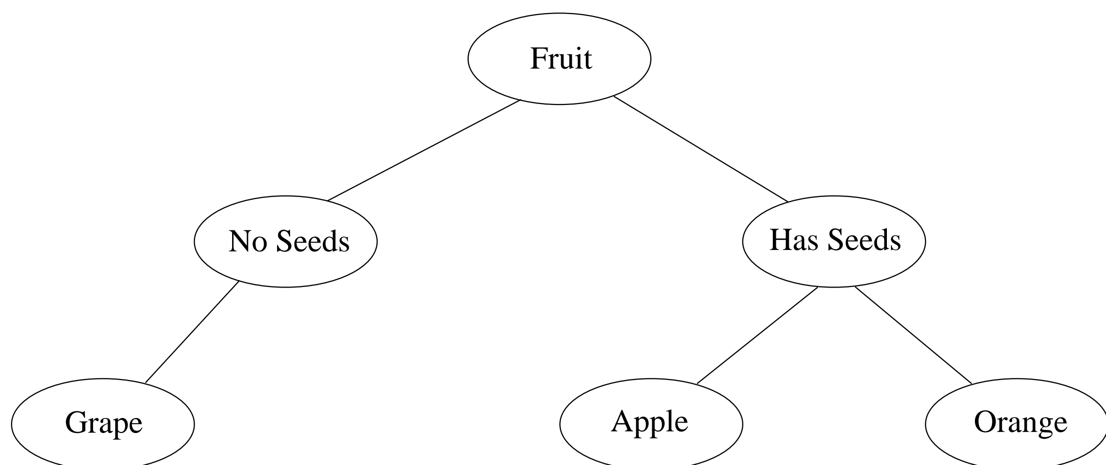
- A *node* (*vertex* or *point*) is a container for data and has relationships with its *predecessor* (if any), and *successors* (if any).
- An *edge* is the relationship of a node to a successor.

- The successors of the root are roots of the *subtrees of the root*. The *parent* of a node (if any) is its predecessor. The *children* of a node (if any) are its successors.
- A node is a *leaf* if it has no children. Otherwise it is an *internal node*
- The *depth/level* of a node is its distance from the root. And the *depth/height* of a tree is the maximum level of nodes in the tree



18.1.3 What is a binary tree?

- A *binary tree* is a rooted tree in which every node has at most two children.
- A child on the left of the node is called the *left child* and a child on the right of the node is called the *right child*
- The *left* (*right*) *subtree* is the subtree containing the left (right) child.



- In a *full binary tree*, every leaf has the same depth, and every nonleaf has two children.

- In a *complete binary tree*, every level except the deepest must contain as many nodes as possible, and at the deepest level, all the nodes are as far left as possible.

18.2 Tree Representations

18.2.1 Array Representation of Complete Binary Trees

- Complete binary trees have a simple representation using arrays.
- In this representation, the root node is stored as the first entry of an array, then the left child of the root node the second entry of the array and the right child of the root the third entry of the array. Then the most left child on the third level goes to the forth place of the array, then the child from left to right goes to the array entry, and level by level. See the example on pages 460 — 461.
- Under this representation the following properties are true:
  - Suppose that the data for a nonroot node appears in entry `[i]` of the array, then the data for its parent is always at location `[(i-1)/2]`.
  - Suppose that the data for a node appears in entry `[i]` of the array, then its children (if they exist) always have their data at these locations: the left child at entry `[2i+1]` and the right child at entry `[2i+2]`.

18.2.2 Representing a Binary Tree with a Node Class

- For a binary tree, we store node information in the object of a node class which has two pointers to its left and right child nodes.
- The node template class is now like

```
template <class Item>
class binary_tree_node {
   public:
     //public member functions
   private:
     Item data_field;
     binary_tree_node<Item>* left_field;
     binary_tree_node<Item>* right_field;
};
```

- See the example on page 463.

18.2.3 Operations on a Binary Tree

- We need to add as many member functions as possible to the `binary_tree_node` class.

- Actually a binary tree is presented by a lot of nodes linked together. We need a function returning the data information of the tree node.

```
template <class Item>
Item& binary_tree_node<Item>::data()
{  return data_field; }
```

- One of important operations is to destroy a binary tree. We need to return the nodes of a tree to the heap. Our function will have one parameter, which is the root pointer of the tree. This function employs a recursive algorithm. That is, before we return the root node to the heap, we should return all the nodes in the left subtree and right subtree.

```
template <class Item>
void binary_tree_node<Item>::tree_clear(binary_tree_node<Item>*& root_ptr)
{
   if (root_ptr != NULL ) {
       tree_clear( root_ptr->left() );
       tree_clear( root_ptr->right() )
       delete root_ptr;
       root_ptr = NULL;
   }
}
```

- Our second function also has a simple recursive implementation. The function copies a tree. The tree can be copied with these three steps
  - Make `l_ptr` point to a copy of the left subtree
  - Make `r_ptr` point to a copy of the right subtree
  - return new tree

```
template <class Item>
binary_tree_node<Item>  binary_tree_node<Item>::
  tree_copy(const binary_tree_node<Item>* root_ptr)
{
   binary_tree_node<Item>* l_ptr;
   binary_tree_node<Item>* r_ptr;
   if (root_ptr == NULL )
       return NULL;
   else {
       l_ptr = tree_copy( root_ptr->left() );
```

```
        r_ptr = tree_copy( root_ptr->right() )
        return new binary_tree_node<Item>(
            root_ptr->data(), l_ptr, r_ptr)
        //using the constructor
    }
}
```

18.3 Traversing Binary Trees

18.3.1 Traversals on Trees

- To *traverse* a tree means to travel through the tree and *visit* each node by performing a task.
- A procedure called when a node is `visited` is called a *visit procedure*.
- There are three different kinds of traversals:
  - **Preorder**
  - **Postorder**
  - **Inorder**

18.3.2 Preorder Traversal

- Visit the root before traversing each subtree.
- Algorithm (for binary tree):

```
if the tree is not empty then
  visit the root
  traverse the left subtree
  traverse the right subtree
```

18.3.3 Postorder Traversal

- Visit the root after traversing each subtree.
- Algorithm (for binary tree):

```
if the tree is not empty then
  traverse the left subtree
  traverse the right subtree
  visit the root
```

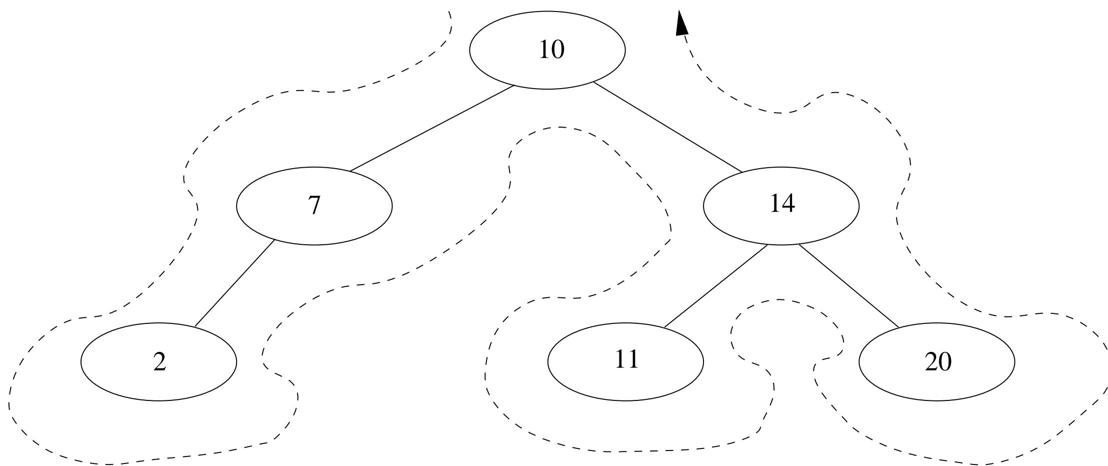
18.3.4 Inorder Traversal

- Visit the root between the traversals of the left & right subtrees.
- Algorithm:

```
if the tree is not empty then
  traverse the left subtree
  visit the root
  traverse the right subtree
```

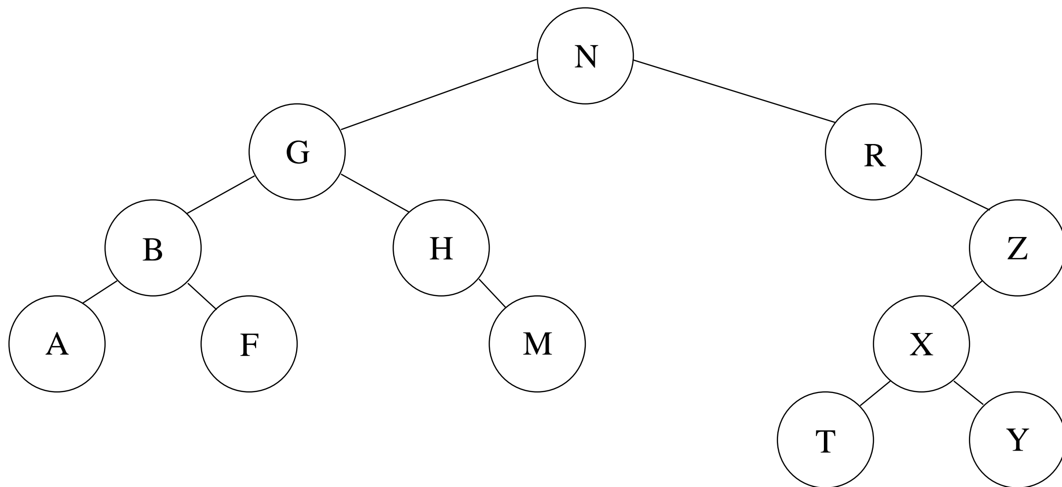
18.3.5 Doing traversals the easy way

- Draw a line around the tree, as such:



- We travel anticlockwise along the line and visit each node as we pass:
  - it on the **left** for preorder
  - it on the **right** for postorder
  - **beneath** it for inorder

18.3.6 Example: Traversing a binary tree



- Preorder: `N G B A F H M R Z X T Y`
- Postorder: `A F B M H G T Y X Z R N`
- Inorder: `A B F G H M N R T X Y Z`

18.3.7 Code Example for Pre-order Traversal

- The visit procedure defined as a function will be transferred to the traversal algorithm as a function pointer parameter.
- The template function definition is given below

```
template <class Process, class BTNode>
void preorder(Process f, BTNode* node_ptr) {
    if (node_ptr != NULL) {
        f(node_ptr->data()) //visit data
        preorder(f, node_ptr->left() );
        preorder(f, node_ptr->right() );
    }
}
```

- `Process` is a type parameter for a function type. You instantiate the type parameter `Process` by any type of function.
- Following the same rule, you can define functions for postorder and inorder traversal algorithms.

Chapter 19

Binary Search Trees and B-Trees

19.1 Binary Search Trees

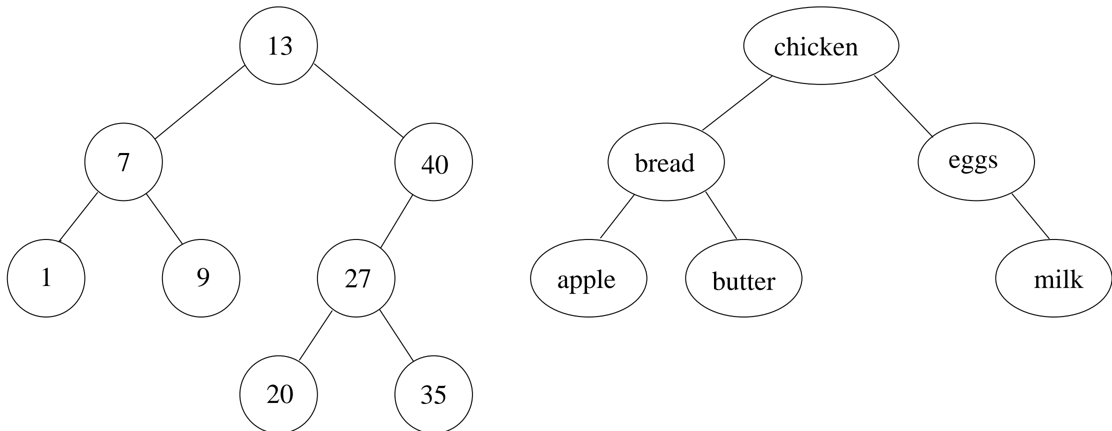
Some of the most significant applications of binary trees arise from storing data in an ordered fashion in what we call a binary search tree.

19.1.1 The definition of Binary Search Trees

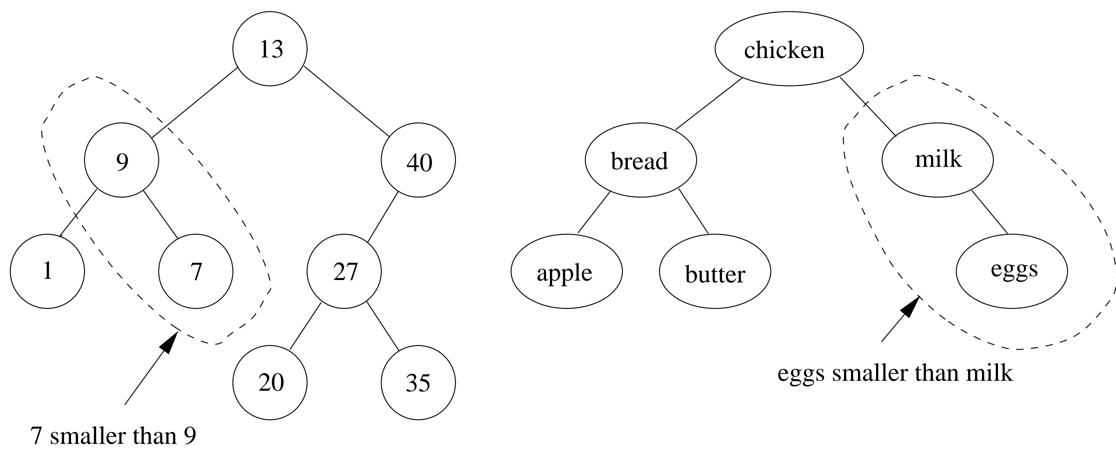
- For each node `node` we must have
    - The entry in `node` is greater than all entries in its left subtree
    - The entry in `node` is less than all entries in its right subtree
    - Both subtrees are binary search trees

19.1.2 Examples

- Two binary search trees



- These are **NOT** binary search trees



19.1.3 Implementation of binary search trees

- Pointer based:

```
template <class Item>
class BSTree {
   public:
        BSTree( );
        BSTree(const Item& source);
        ~BSTree( );
        bool erase_one(const Item& target);
        size_type erase(const Item& target);
        void insert(const Item& entry);
        void operator += (const BSTree& addend);
        void operator =(const BSTree& source);
        size_type size() const;
        size_type count(const Item& target);
         void inorder(void (*)(Item&), binary_tree_node<Item>*);

   private:
         binary_tree_node *root_ptr;
};
```

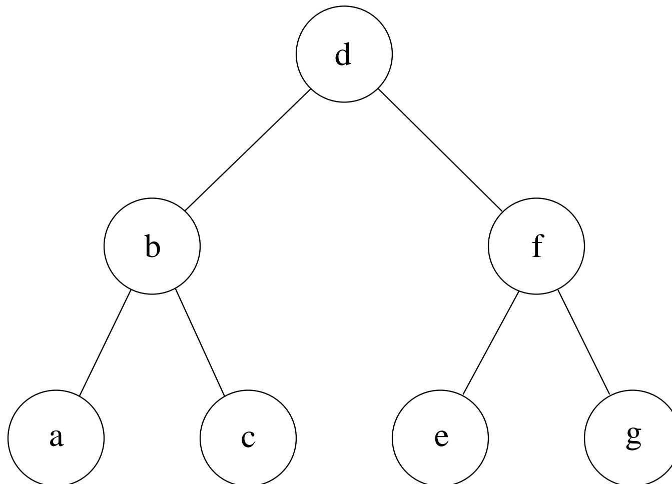
- Array based: not covered

19.2 Main Operations on Binary Search Trees

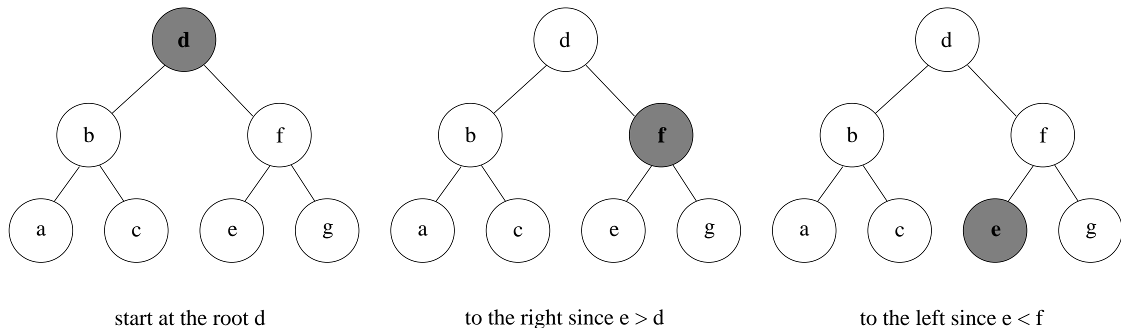
19.2.1 Searching for an entry

- We start at the root

- Three cases:
  1. The root is the node containing the entry we are looking for
  2. The entry in the root node is larger than the target entry, then search *left* subtree
  3. The entry in the root node is smaller than the target entry, then search *right* subtree

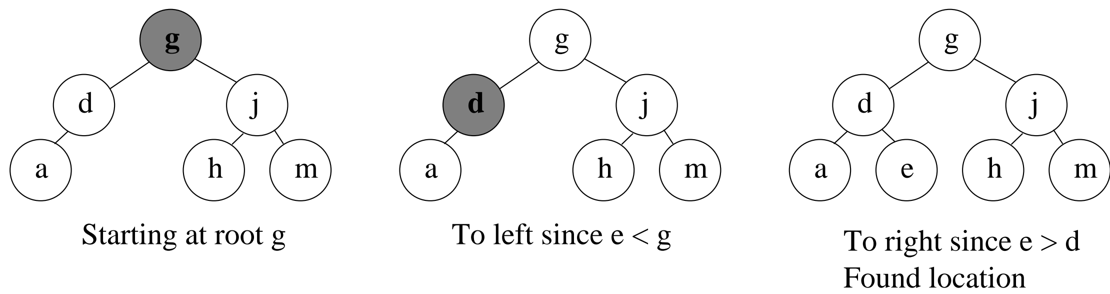


19.2.2 Searching for the Node Containing `entry`



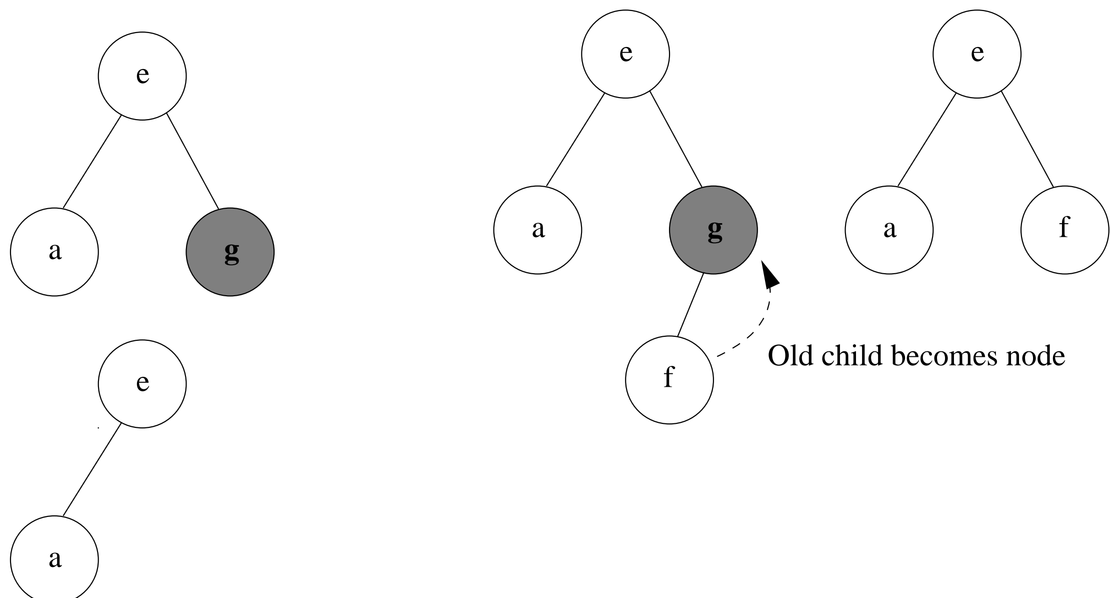
19.2.3 Inserting an Element

- Two high level steps:
  - Find correct location for new node
  - Insert node at that location
- Finding location is like searching for an entry
- Example

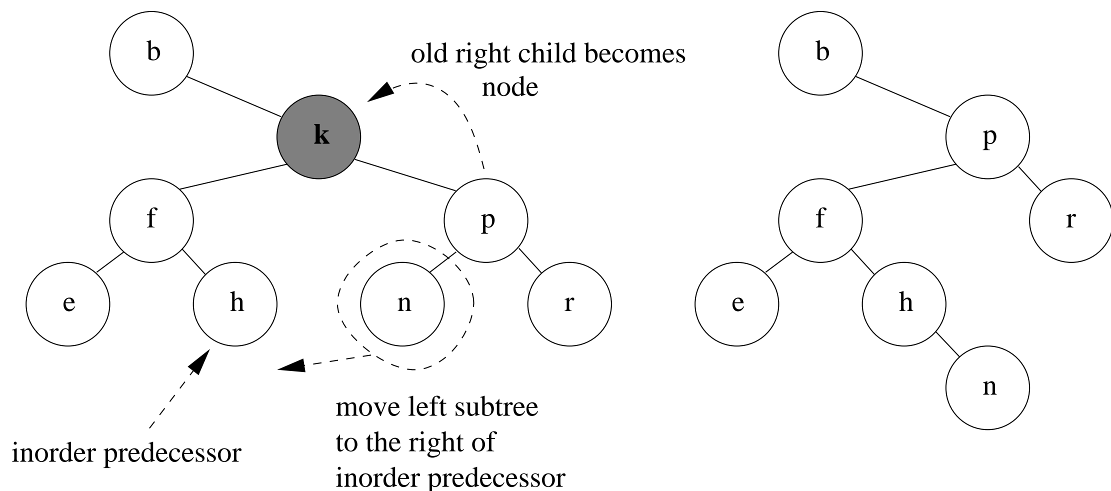


19.2.4 Deleting an Element

- There are three cases:
  - No children: *easy*
  - One child: *easy*
  - Two children: *difficult*
- Deleting a node with zero or one children



- Deleting a node with two children



- This method tends to produce higher trees which is undesirable, and causes the operations to take longer.

19.3 B-Trees

19.3.1 The Problem of Unbalanced Trees

- Binary search tree can be used to implement a bag class. However the efficiency of binary search trees can go awry.
- Suppose we add numbers in increasing order into a binary search tree, then we will get an unbalanced tree in the sense that the levels of the tree are only sparsely filled. See Figure 11.3 of the text.
- It will take long time to look up the "maximal" item in such an unbalanced binary search tree.
- To make algorithm efficient on binary search trees we need to reduce the depth of the trees.

19.3.2 The B-Tree Rules

- A B-tree is a special kind of tree, similar to a binary search tree, where each node holds entries of some type. But a B-tree is not a binary search tree, even not a binary tree because B-tree nodes have many more than two children.
- Another important property of B-trees is that each node contains more than just a single entry. Every B-tree depends on a positive constant integer called `MINIMUM`. This constant determines how many entries are held in a single node.
- *B-tree Rule 1*: The root may have as few as one entry (or even no entries if it also has no children); every other node has at least `MINIMUM` entries.
- *B-tree Rule 2*: The maximum number of entries in a node is twice the value of `MINIMUM`.

- *B-tree Rule 3*: The entries of each B-tree node are stored in a partialy filled array, sorted from the smallest entry (at index 0) to the largest entry (at the final used position of the array).
- *B-tree Rule 4*: The number of subtrees below a nonleaf node is always one more than the number of entries in the node.
- *B-tree Rule 5*: For any nonleaf node: (a) An entry at index i is greater than all the entries in subtree number i of the node, and (b) Any entry at index i is less than all the entries in subtree number $i + 1$ of the node.
- *B-tree Rule 6*: Every leaf in a B-tree has the same depth.
- B-tree example

19.4 The Set Class for B-tree Nodes

19.4.1 The Set ADT for B-trees

- Each node of a B-tree has at least `MINIMUM` entries and less than `2*MINIMUM+1` entries. We will define a set class for each B-tree node.
- For each B-tree node, we use `data_count` to denote the actual number of the entries and `child_count` to denote the actual number of children (subtrees).
- Here is the recipe of the set template class

```
template <class Item>
class set {
   public:
      //set member functions
   private:
      static const std::size_t MINIMUM = 200;
      static const std::size_t MAXIMUM = 2 * MINIMUM;
      std::size_t data_count;
      Item data[MAXIMUM+1];
      std::size_t child_count;
      set *subset[MAXIMUM+2];
};
```

- From the above structure we can see that the subtrees of the B-tree root node (set) are pointed to by pointers stored in the `subset` array of type `set`, that is, each array entry `subset[i]` is a pointer to another `set` object.

19.4.2 The Example

19.5 B-tree Operations

19.5.1 Searching for an Item in a B-tree

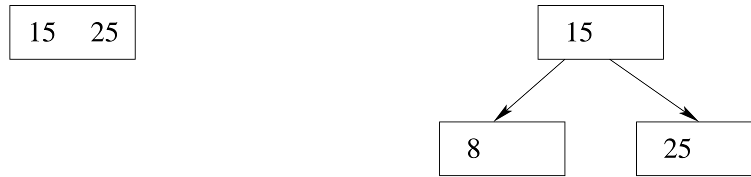
- In the implementation of the set class for a B-tree, the member function `count` determines whether an item called `target` appears in the set. Searching for an item in a B-tree follows the same idea as searching a binary searh tree.
- Start with the entire B-tree, checking to see whether the target is in the root. If the target does appear in the root, then the search is done. Or the target is not in the root, and the root has no children. In this case the work is also done.
- If the target is not in the root and there are subtrees below. In this case, there is only one possible subtree where the target can appear, so the algorithm makes a recursive call to search that one subtree for the target.
- The pseudocode is

```
set a variable i to the first index such as that data[i] is not
less than the target. If there is no such index, then set i
to data_count, meaning that all of the entries are less than
the target

if (we found the target at data[i]
    return 1;
else if (the root has no children)
    return 0;
else
    return subset[i]->count(target);
```

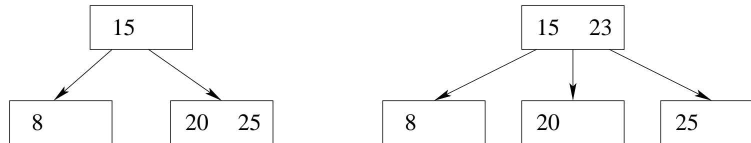
- See the example from page 533 of the text.

19.5.2 Inserting an Item into a B-tree

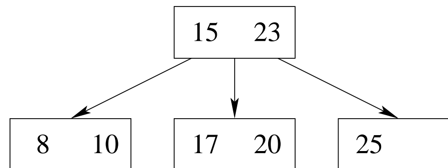
- A B-tree maintains its "bushy" characteristic by growing at the root instead of at the leaves.
- To illustrate how an insertion works, suppose we are developing a B-tree within `MINIMUM=1`, so that each node has a maximum of two data items and three subtrees.
- Suppose we wish to insert the following values into the tree: 25, 15, 8, 20, 23, 10, 17, and 12.
- Because the root can accept two values, then at the beginning the root node contains items 15 and 25 (`data[0]=15` and `data[1]=25`).



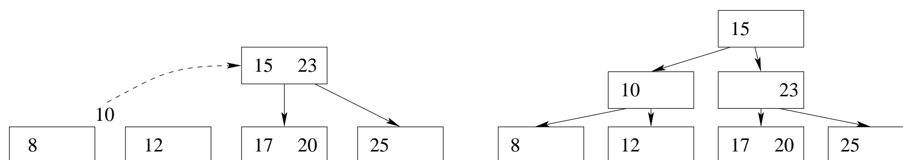
- There is no room for the next item 8. To perform the insertion, we split the node in two and form a new root with these nodes as children. The middle of the sorted value 8, 15 and 25 goes into the root.
- Because the next value 20 is greater than 15, it goes in the rightmost child.



- There is no room for the next value 23 which should also be in the rightmost subtree. We sort the values, 20, 23, 25, and , as before, divide the node. The middle value 23 goes to the root, while 20 and 25 proceed to separate nodes.
- In the next step, it is easy to insert the items 10 and 17.



- The value 1 also needs to go into the left child, but there is no room. Thus we sort 8, 10, 12, place 8 and 12 in their own nodes, and send the middle value to the parent. However the parent (the root) already is full. Consequently, we repeat the process. We sort, 10, 15, and 23, move the middle value 15 to a new root and place 10 and 23 in leftmost and rightmost children, respectively. Instead of propagating downward as with binary search trees, B-trees propagate upward.



19.5.3 Removing an Item from a B-tree

- In the deletion operation, the algorithm removes an entry from the B-tree. Most of the work will be accomplished with a procedure called “loose removal” by which we allow to leave a root of the wholde tree with zero entries or it might leave the root of an internal subtree with fewer than `MINIMUM` entries.

- The algorithm is fully described in the textbook. Before you read the text, please try yourself to see if you can work out the algorithm

Chapter 20

Quadratic Sorting

One of the most common applications in computer science is **sorting**, the process through which data are arranged according to their values. We discuss six sorting algorithms.

20.1 Concepts Related to Sorting

20.1.1 What shall we learn?

- To learn how various sorting methods work
  - Description and Explanation
  - Algorithm
  - Implementation
- To learn the efficiency of various sorting methods and about limitations on the best possible methods
- To learn how to select a sorting technique that is well-matched to the characteristic of a problem you need to solve

  Lesson: No method is better under all circumstances

20.1.2 When Analyzing a Sorting Method

- The time complexity of the algorithm
- The amount data movement required
- The average-case and worst-case performance are of interest

20.1.3 Ordered Array

- An *ordered array* is an array in which each entry contains a key, such that the keys are in order.
- That is, if entry `i` comes before entry `j` in the array, then the key of entry `i` is *less than* or *equal to* the key of entry `j`.

- Two different elements may have the same key.

20.1.4 Classification of Sorting

- In-Place Sort and External Sort
  - An *in-place sort* of an array occurs within the array and uses no external storage or other arrays
  - In-place sorts are more efficient in space utilization
  - An external sort uses primary memory for the data currently being sorted and secondary storage for any data that will not fit in primary memory.
- Stable Sorts
  - A sort is *stable* if it preserves the ordering of elements with the same key.
  - i.e. If elements `e1` and `e2` have the same key, and `e1` appears earlier than `e2` before sorting, then `e1` is located before `e2` after sorting.
  - Example:
    - Suppose we have an array of names and addresses that are already sorted by name.
    - We want to have an ordering of these people by city.
    - We want to preserve the alphabetical ordering for each city.
    - We must use a stable sorting algorithm.

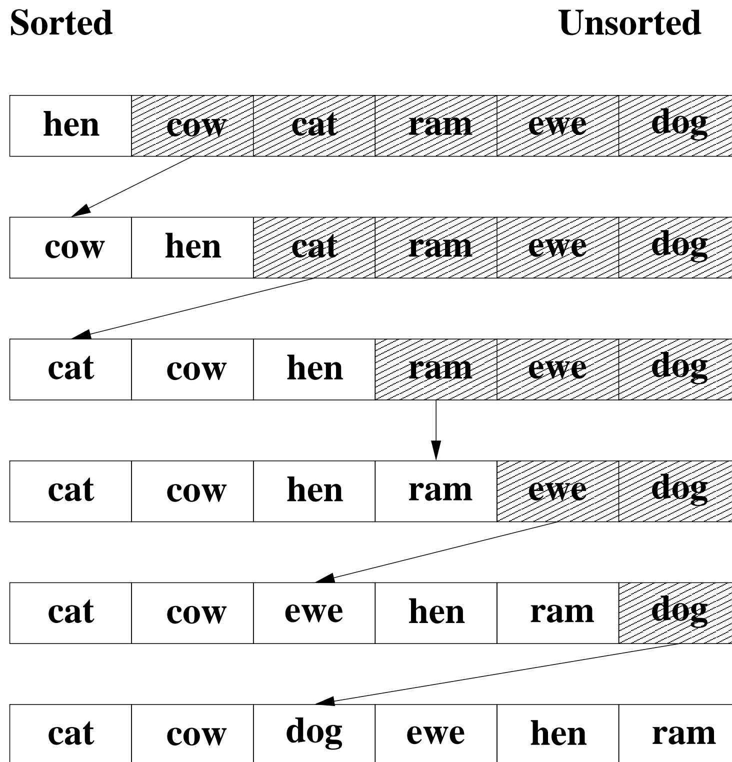
20.2 Insertion Sort

We talk about the description of method, illustration of method, algorithm, possible C++ function and its characteristics.

20.2.1 Description of Insertion Sort

- Straightforward method that is often useful for small collections of data ($n \leq 100$).
- The general idea of insertion sort is for each element, find the slot where it belongs.
- We have a sub-array of sorted elements.
- At each step of the sort, we insert the next unsorted element into the sorted sub-array

20.2.2 Illustration of Insertion Sort



20.2.3 Insertion Sort Algorithm

We take each element $a[i]$, one at a time, from i equal 1 to equal $n-1$. Before insertion of the value of $a[i]$, the subarray from $a[0]$ to $a[i-1]$ is sorted, and the remainder of the array is not. After insertion, $a[0..i]$ us correctly ordered while the subarray with elements $a[i+1], a[i+2], ..., a[n-1]$ is unsorted.

Pre:

a is an array with n elements

Post:

$a[0..n-1]$ has been sorted

Algorithm:

```
for i from 1 to n-1
  temp = a[i]
  loc = i
  while ((loc > 0) and (a[loc-1] > temp))
    a[loc] = a[loc-1]
    loc = loc-1
  a[loc] = temp
```

20.2.4 Insertion Sort Function

- We should document any assumption about template arguments.

```
// Pre: > is defined for type T
template <class T>
void insertSort(T *a, size_t n) {
  size_t i;
  size_t loc;
  T    temp;

  for(i = 1; i < n; ++i) {
      temp = a[i];
      loc  = i;
      while(loc && (a[loc - 1] > temp)) {
         a[loc] = a[loc - 1];
         --loc;
      }
      a[loc] = temp;
   }
}
```

20.2.5 Characteristics of `insertSort`

In–Place Sort	Y
Stable Algorithm	Y
Recursive Algorithm	N

	Average Case	Best Case	Worst Case
Time Complexity	O(n^2)	O(n)	O(n^2)
Circumstances		sorted	reverse sorted
Number of Moves	O(n^2)	O(n)	O(n^2)

The outer `for` loop is executed $n-1$ times, from 1 to $n-1$. On the i-th iteration to find and prepare the location for `temp`, the value of the i-th array element, we execute the `while` loop. If $a[i]$ is already in its proper location, the `while` loop does not execute. Thus our best situation is where the data is already sorted. However, in the worst case the array is in descending order, then the insertion sort to place the array in ascending order requires a total number of $1+2+...+(n-1)$ iteration of the `while` loop. Now you can find the average number of times.

One of disadvantage of insertion sort of an array is the amount of movement of data. In an array of length records, those reassignments can be quite time-consuming.

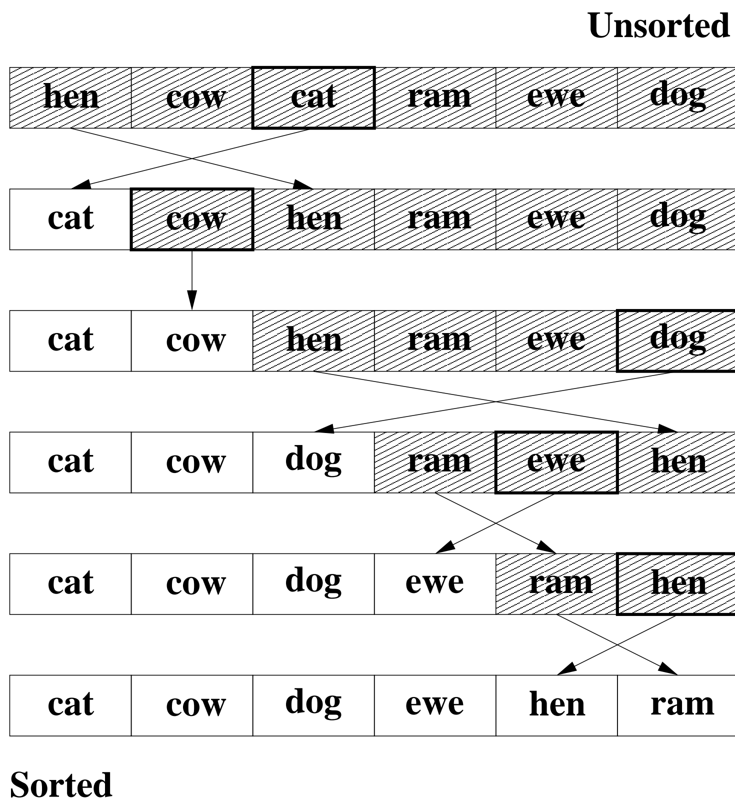
20.3 Selection Sort

We discuss the description of method, illustration of method, algorithm, C++ Function and its characteristics.

20.3.1 Description of Selection Sort

- For each slot of the array, we find the element that belongs there.
- The idea is basically opposite to that for insertion sort.
    - Insertion Sort: Find slots for elements
    - Selection Sort: Find elements for slots
- Good for big elements with small keys

20.3.2 Illustration of Selection Sort



20.3.3 Selection Sort Algorithm

Pre:

a is an array with n elements

Post:

$a[0..n-1]$ has been sorted

Algorithm:

```
for i from 0 to n-2 do
  minIndex = indexOfMin(a, i, n)
  swap(a[i], a[minIndex])
```

20.3.4 `indexOfMin` Algorithm

Pre:

a is an array with n elements.

i is an index and $i < n$.

Post:

The function returns the index of the minimum element in the subarray $a[i..(n-1)]$.

Algorithm:

```
minIndex = i
for j from i+1 to n-1 do
  if a[j] < a[minIndex] then
    minIndex = j
return minIndex
```

20.3.5 Selection Sort Function

```
// Pre: > is defined for type T

template <class T>
void selSort(T *a, size_t n) {
   size_t  i;
   size_t  minIndex;

   for(i = 0; i < n-1; ++i) {
      minIndex = indexOfMin(a, i, n);  //must run
      swap(a[i], a[minIndex]);
   }
}

// Pre: > is defined for type T
template <class T>
size_t indexOfMin(T *a, size_t i, size_t n) {
   size_t j;
   size_t minIndex = i;
```

```
    for(j = i + 1; j < n; ++j) {
        if(a[j] < a[minIndex])
            minIndex = j;
    }
    return minIndex;
}

template <class T>
void swap(T &x, T &y) {
  T temp = x;
  x = y;
  y = temp;
}
```

20.3.6 Characteristics of Selection Sort

In–Place Sort	Y
Stable Algorithm	N
Recursive Algorithm	N

	Average Case	Best Case	Worst Case
Time Complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$
Number of Moves	O(n)	O(n)	O(n)

To discover the complexity, we notice that the order loop is a `for` loop that always executes $n-1$ times. The inner `for` loop, in `indexOfMin` function, is performed $(n-1)-(i+1)+1 = n-i-1$ times. The total number of execution of the `if` statement in the inner loop is

$$(n-1) + ... + 3 + 2 + 1 = \frac{1}{2}n^2 - \frac{1}{2}n$$

Regardless of the ordering of the data, the number of comparisons in a selection sort is $O(n^2)$.

20.4 Comparing Insertion and Selection Sort

- Insertion Sort:
  - One major disadvantage: an entry may be moved many times
  - Stable algorithm, does not alter ordering of elements with same key
- Selection Sort:
  - Insensitive to original ordering, slower than insertion sort if array is almost sorted

- The worst-case as good as the best-case
- An entry never gets moved if it is originally in its final position; at most n swaps required
- Unstable algorithm, may alter ordering of elements with same key

Chapter 21

Recursive Sorting

21.1 Mergesort

We learn the idea of divide and conquer. The mergesort method is faster than Insertion and Selection Sort, but not often used due to the time spent on combining the two subarrays.

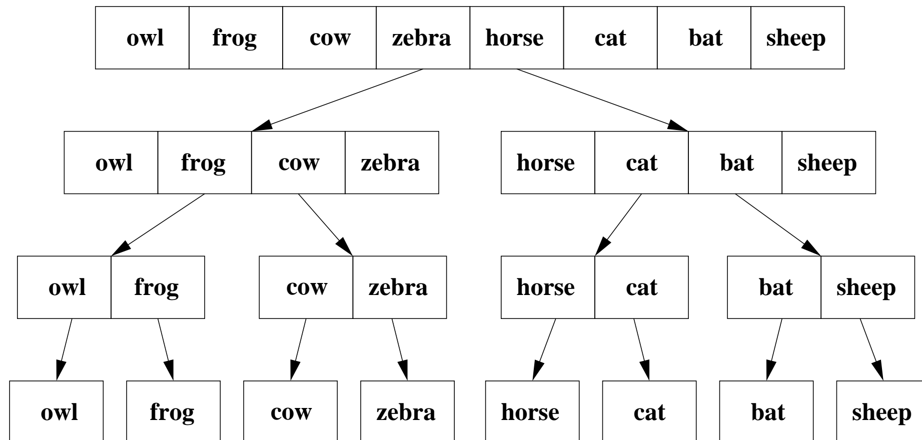
21.1.1 Divide and Conquer for Mergesort

- Divide the array into two subarrays — Easy Step
- Sort each individual array — Recursive Step
- Combine the two subarrays — Time Consuming Step

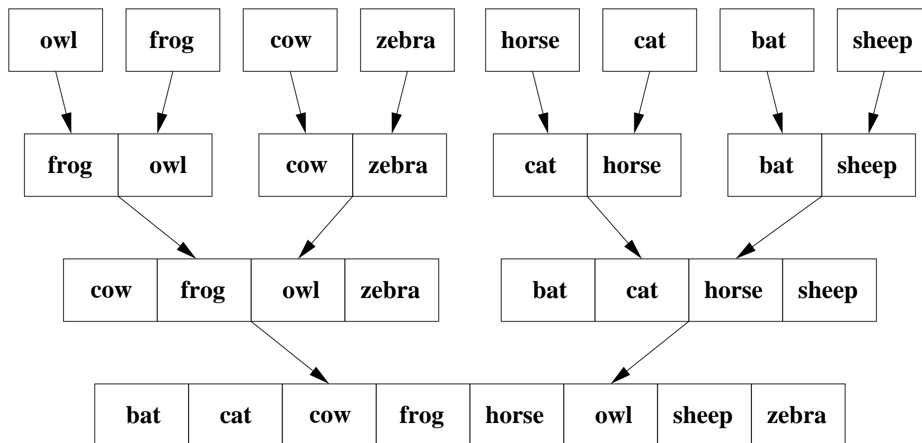
21.1.2 Three Stages of Mergesort

1. Divide the array into subarrays:
    - Choose the middle element:
        - `mid = (first+last)/2`
    - The left subarray is `a[first..mid]`
    - The right subarray is `a[(mid+1)..last]`
2. Sort the subarrays
    - Recursive call to `Mergesort`
3. Merge the two subarrays
    - Merge subarrays into a temporary array
    - Copy temporary array back into original array

21.1.3 Recursive calls to Mergesort



21.1.4 Merges of adjacent, sorted subarrays



21.2 Algorithm

21.2.1 `Mergesort(a, first, last)` Algorithm

Pre:

`a` is an array with values for indices from `first` to `last`

Post:

`a[first..last]` is sorted

Algorithm:

```
if first < last then
  mid = (first + last)/2
  Mergesort(a, first, mid)
  Mergesort(a, mid+1, last)
  merge(a, first, mid, last)
```

21.2.2 Merging two subarrays

- Create space for `temp` array
- While there still elements to consider in both subarrays
  - Get the smallest element
  - Add it to the `temp` array
  - Go to next element in that subarray
  - Go to next element in the `temp` array
- Add remaining elements to `temp` array
- Copy `temp` array back into original array

21.2.3 `Merge(a, first, mid, last)` Algorithm

Pre:

`a` is an array in which subarrays `a[first..mid]` and `a[(mid+1)..last]` are sorted.

`first`, `mid`, and `last` are indices of `a`, where `first` $\leq$ `mid` $\leq$ `last`.

Post:

`a[first..last]` is sorted

Algorithm:

```
ndx1  = first
last1 = mid
ndx2  = mid+1
last2 = last
i     = 0

while ((ndx1 <= last1) and (ndx2 <= last2))
  if a[ndx1] < a[ndx2] then
    temp[i++] = a[ndx1++]
  else
    temp[i++] = a[ndx2++]

while (ndx1 <= last1)
  temp[i++] = a[ndx1++]

while (ndx2 <= last2)
  temp[i++] = a[ndx2++]

i = 0
for j from first to last
  a[j] = temp[i++]
```

21.2.4 Mergesort Functions

```
template <class T>
void MergeSort(T a[], size_t first, size_t last) {
   if(first < last) {
      size_t mid = (first + last) / 2;
      MergeSort(a, first, mid);
      MergeSort(a, mid + 1, last);
      merge(a, first, mid, last);
   }
}

template <class T>
void merge(T a[], size_t first, size_t mid, size_t last) {
   size_t ndx1   = first;
   size_t last1  = mid;
   size_t ndx2   = mid + 1;
   size_t last2  = last;
   size_t i      = 0;
   size_t j;
   T* temp = new T[last - first + 1];
   if(!temp) {
      cerr << "Error allocating memory.";
      exit(1);
   }

  while(ndx1 <= last1 && ndx2 <= last2) {
      if(a[ndx1] <= a[ndx2])
         temp[i++] = a[ndx1++];
      else
         temp[i++] = a[ndx2++];
   }
   while(ndx1 <= last1)
      temp[i++] = a[ndx1++];
   while(ndx2 <= last2)
      temp[i++] = a[ndx2++];

   i = 0;
   for(j = first; j <= last; j++)
      a[j] = temp[i++];
```

```
    delete [] temp;
}
```

21.2.5 Mergesort Characteristics

In–Place Sort	N
Stable Algorithm	Y
Recursive Algorithm	Y

	Average Case	**Best Case**	**Worst Case**
Time Complexity	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

21.3 Quicksort

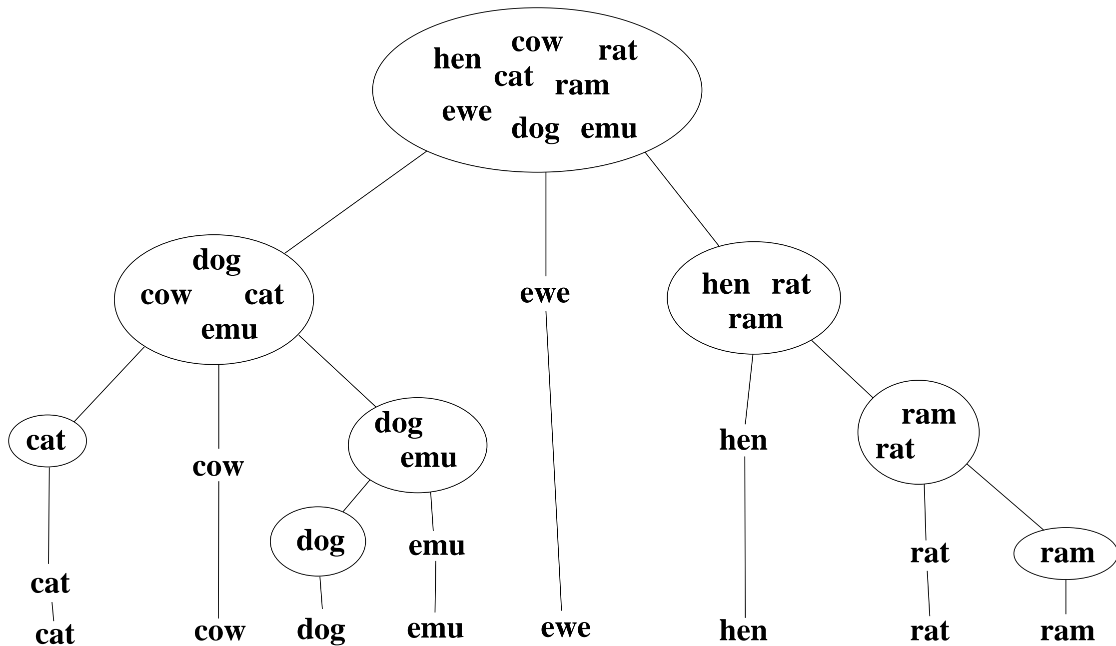
We discuss the idea of quicksort, how it works for both arrays & linked lists, How to choose the pivot, and how to implement in C++ code.

21.3.1 Quicksort Method

- The method is presented by **C. A. R. Hoare** in 1962. It is a recursive algorithm and far better performance on average than insertion & selection sort It is arguably the best sorting method when applied to arrays in random order. It uses the tqchnique of divide & conquer.
- The method is to choose an element as the `pivot`, then to partition the array into two subarrays:
    - Elements *less* than the pivot
    - Elements *greater* than the pivot

  and recursively to repeat the process on the subarrays.

21.3.2 Illustration of Method



21.3.3 `Quicksort(a, first, last)` Algorithm

Pre:

a is an array

$first$ and $last$ are indices within the range of a

Post:

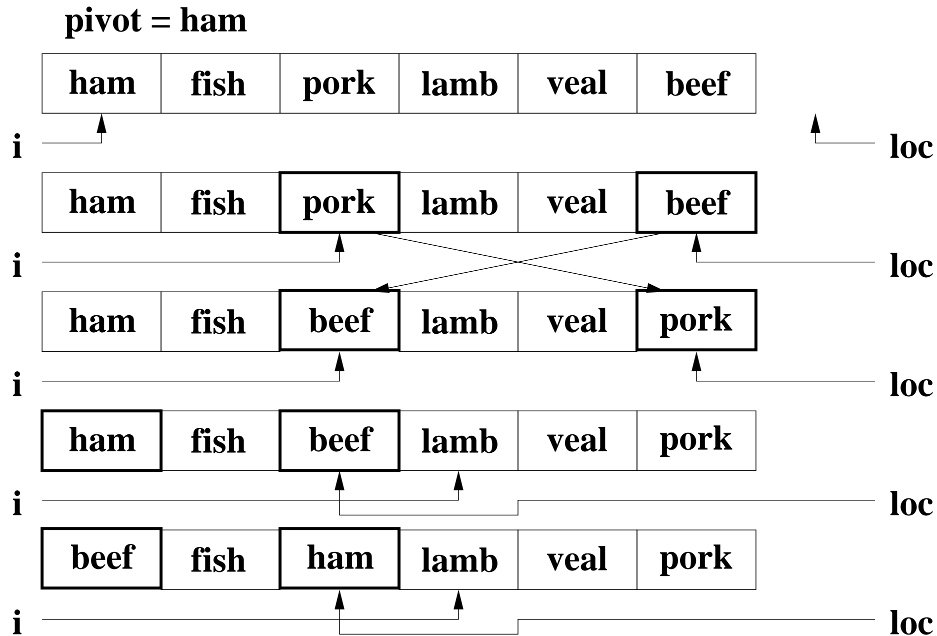
$a[first..last]$ has been sorted

```
if first < last then
  partition(a, first, last, loc)
  Quicksort(a, first, loc - 1)
  Quicksort(a, loc + 1, last)
```

21.3.4 `partition`: the basic idea

- Choose first element as pivot
- Going from the left, find an entry which belongs on the right of the pivot ($\geq$ pivot)
- Going from the right, find an entry which belongs on the left of the pivot ($\leq$ pivot)
- Swap these entries
- Repeat until we meet in the middle

21.3.5 Illustration of partition



21.3.6 `partition(a, first, last, loc)` Algorithm

Pre:

a is an array

$first$ and $last$ are are indices within the range of a and $first < last$

Post:

$a[first]$ is chosen as the $pivot$

$a[first..(loc-1)]$ contains elements less than or equal to the $pivot$

$a[loc]$ contains the $pivot$

$a[(loc+1)..last]$ contains elements greater than or equal to the $pivot$

loc is the location of the $pivot$ in the partitioned array

Algorithm:

```
i = first
loc = last + 1
pivot = a[first]

while (i < loc)
  do
    i = i + 1
  while ((a[i] < pivot) and (i < last))
```

```
  do
    loc = loc - 1
  while (a[loc] > pivot)

  if (i < loc)
    swap(a[i], a[loc])

swap(a[first], a[loc])
```

21.3.7 Quicksort: C++ Style

```
template <class T>
void quickSort(T *a, size_t first, size_t last) {
   if(first < last) {
      size_t loc = partition(a, first, last);
      quickSort(a, first, loc - 1);
      quickSort(a, loc + 1, last);
   }
}
```

21.3.8 The Partition function in C++

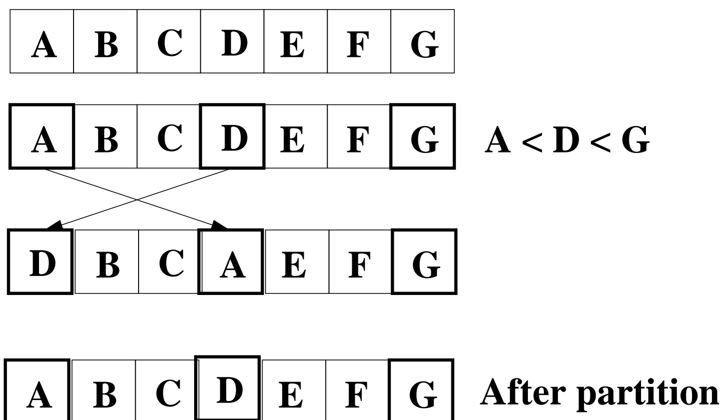
```
template <class T>
size_t partition(T *a, size_t first, size_t last) {
   size_t i      = first;
   size_t loc    = last + 1;
   T pivot    = a[first];

   while (i < loc) {
      do {
         ++i;
      } while ((a[i] < pivot) && (i < last));
      do {
         --loc;
      } while (a[loc] > pivot);

      if(i < loc)
         swap(a[i], a[loc]);
   }
   swap(a[first], a[loc]);
```

```
    return loc;
}
```

21.3.9 Worst Case & median-of-three

- The worst case occurs when the array is either sorted or reverse sorted
- In both cases, the size of the subarray to be sorted at each recursive step is only reduced by one.
- The **median-of-three** method avoids the poor performance of quicksort when the elements are sorted.
- Instead of chosing $a[first]$ as a pivot, the *median* of the $first$, $last$ and middle elements is computed
- It is then swapped into $a[first]$ before the partition occurs

21.3.10 Example: median-of-three



21.3.11 Quicksort Characteristics

In–Place Sort	Y
Stable Algorithm	N
Recursive Algorithm	Y

	Average Case	Best Case	Worst Case
Time Complexity	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n^2)$
Circumstances			sorted or reverse sorted

Chapter 22

Heap Sort

In this lecture we discuss Heapsort:

- What is a complete binary tree?
- What is a heap?
- How do we form a heap?
- Efficiency
- `comparison` functions

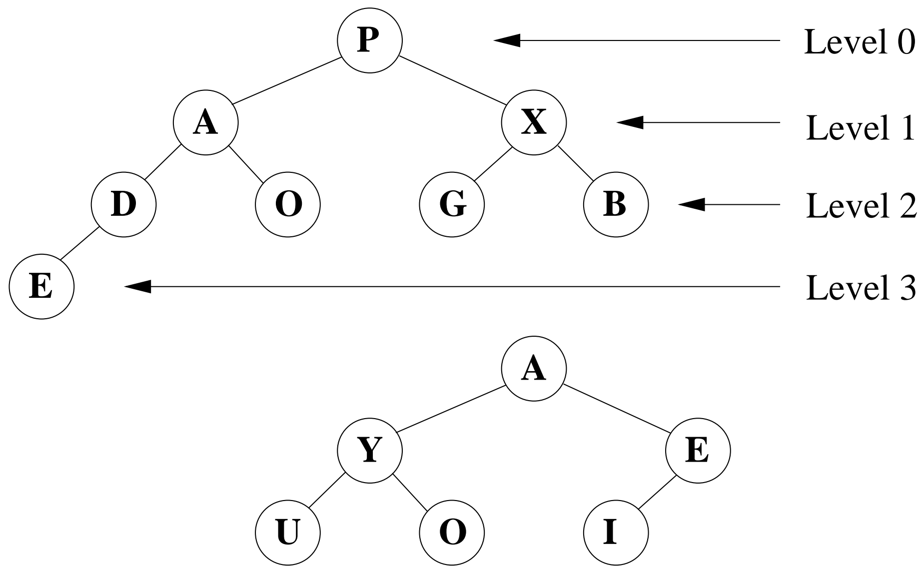
22.1 Complete Binary Trees

22.1.1 Definition of Complete Binary Trees

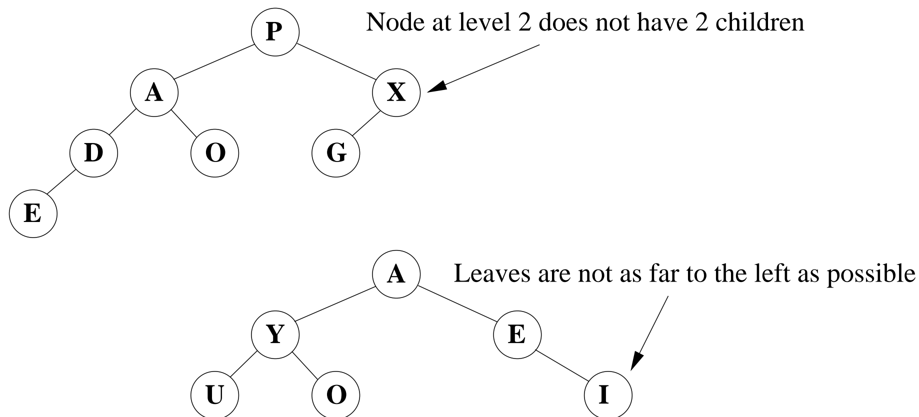
- A *complete binary tree* is a binary tree where:

  1. All of its leaves are on level $n-1$ or level n
  2. On levels 1 through $n-2$, every node has exactly two children
  3. On level n, the leaves are as far to the left as possible

22.1.2 These are Complete Binary Trees

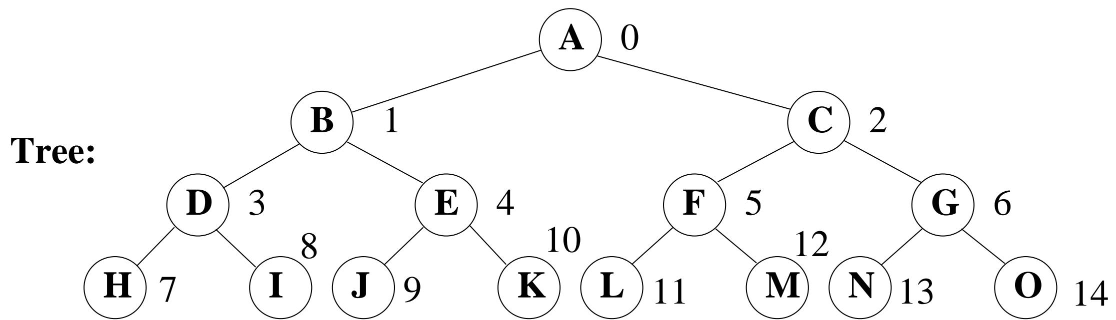


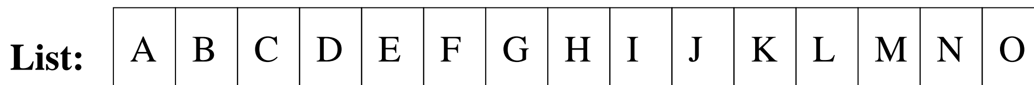
22.1.3 These are *not* Complete Binary Trees



22.1.4 Complete Binary Trees & Arrays

Store the tree *level-by-level*:





If a node has index k **then its left child has index** 2k+1 **and its right child has index** 2k+2

22.2 Heaps

22.2.1 Definition of Heaps

- Definition:

  A *heap* is a complete binary tree, where the values stored in a node is greater than or equal to the values of its children.

- The heapsort function works on a list not on a tree, but it is always convenient to draw a tree to show the hierarchical relationships between the entries of the list.

- For Arrays:

  The key at index k is at least as large as the keys in positions $2k + 1$ and $2k + 2$.

22.2.2 Rough Algorithm of Heapsort

1. Build a heap from the given list:

   (a) Start with an empty heap

   (b) Add elements onto heap one by one, swapping where necessary

2. Sort the resulting heap:

   (a) Swap the first element with the last element of the heap

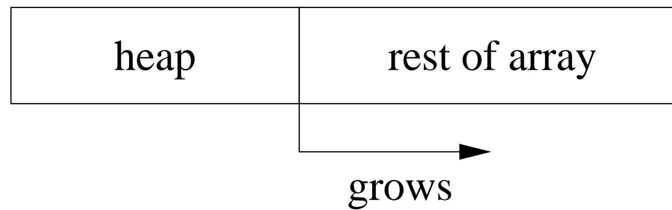
   (b) Reduce the size of the heap by one

   (c) Remake the heap

   (d) Repeat while there are at least two elements left on the heap

22.2.3 Constructing a Heap

- Start with empty heap
- We add the elements one by one
- Find correct location for new element
  - If the new element is larger than its parent, swap them
  - Repeat process until we reach the root, or element is smaller than its parent



22.3 Algorithms

22.3.1 `PlaceInHeap` Algorithm

```
while ((not a heap) and (the child is not at the root)) do:
  find the parent
  if parent and child are out of order then
    swap parent and child
    move the child marker up the tree one level
  else
    we have a heap
```

22.3.2 Refined Algorithm: `PlaceInHeap(h, childNX)`

Pre:

h is an array such that $h[0..(childNX - 1)]$ is a heap

$childNX$ is an index of h

Post:

$h[0..childNX]$ is a heap

```
parentNX = (childNX-1)/2
heap = FALSE

while ((not heap) and (childNX > 0)) do
  if (h[parentNX] < h[childNX])
    swap(h[parentNX], h[childNX])
```

```
    childNX = parentNX
    parentNX = (childNX-1)/2
  else
    heap = TRUE
  end if
end while
```

22.3.3 C++ Code

```
template <class T>
void PlaceInHeap(T h[], size_t childNX) {
   boolean_t  heap        = FALSE;
   size_t     parentNX    = (childNX - 1) / 2;

   while (!heap && childNX) {
      if(h[parentNX] < h[childNX]) {
         swap(h[parentNX], h[childNX]);
         childNX = parentNX;
         if(childNX)
            parentNX = (childNX - 1) / 2;
      }
      else
         heap = TRUE;
   }
}
```

22.3.4 `BuildHeap(h, n)` Algorithm

Pre:

h is an array storing a complete binary tree

n is the number of elements in h

Post:

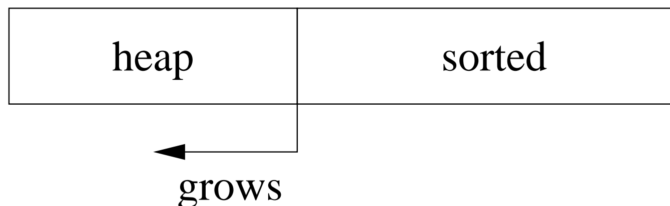
h is an array storing a heap

```
for childNX from 1 to n-1 do
  PlaceInHeap(h, childNX)
```

22.3.5 Sorting the heap

- The largest element will be at the top of the heap
- Swap it to the end of the array (which is where it belongs)

- Reheap the remaining n-1 elements
  - The only element out of order will be the top element
  - While it is not in the correct position, swap it with its largest child
- Repeat process until the array is sorted



22.3.6 Reheap Algorithm

Reheap(h, sortNX):

```
start parent at the root
give childNX the index of the left child
while ((not heap) and (the left child exists))
  if (the right child exists)
    give childNX the index of the larger child
    if (the larger child is greater than the parent)
      swap parent and child
      move parent down one level
      give childNX the index of its left child
    else
      we have a heap
```

22.3.7 C++ code

```
template <class T>
void Reheap(T h[], size_t sortNX)
{
   size_t       parentNX = 0;
   size_t       childNX  = (2 * parentNX) + 1;
   boolean_t heap      = FALSE;
   size_t       RchildNX;

   while (!heap && (childNX < sortNX)) {
      RchildNX = childNX + 1;
      if((RchildNX < sortNX) &&
         (h[RchildNX] > h[childNX]))
        childNX = RchildNX;
```

```
      if(h[childNX] > h[parentNX]) {
         swap(h[childNX], h[parentNX]);
         parentNX = childNX;
         childNX  = (2 * parentNX) + 1;
      }
      else
         heap = TRUE;
   }
}
```

22.3.8 `BuildSortTree(h, n)` Algorithm

Pre:

h is an array that stores a heap

n is the number of elements in h

Post:

h is a sorted array

```
for sortNX from n-1 down to 1 do
  swap(h[0], h[sortNX])
  Reheap(h, sortNX)
end for
```

22.3.9 `Heapsort(h, n)` Algorithm

Pre:

h is an array

n is the number of elements in h

Post:

h is a sorted array

```
BuildHeap(h, n)
BuildSortTree(h, n)
```

22.3.10 C++ Code

```
template <class T>
void BuildHeap(T h[], size_t n) {
   for(size_t childNX = 1; childNX < n; ++childNX)
```

```
        PlaceInHeap(h, childNX);
}

template <class T>
void BuildSortTree(T h[], size_t n) {
    for(size_t sortNX = n - 1; sortNX; --sortNX) {
        swap(h[0], h[sortNX]);
        Reheap(h, sortNX);
    }
}

template <class T>
void HeapSort(T h[], size_t n) {
    BuildHeap(h, n);
    BuildSortTree(h, n);
}
```

22.4 Characteristics of Heapsort

In–Place Sort	Y
Stable Algorithm	N
Recursive Algorithm	N

	Average Case	Best Case	Worst Case
Time Complexity	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$

22.4.1 Analysis of Heapsort

- The worst case:

	Comparisons	Assignments
Quicksort	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log_2 n)$	$O(n \log_2 n)$

- On average, Heapsort takes twice as long as Quicksort
- For small n, Heapsort is inefficient

Chapter 23

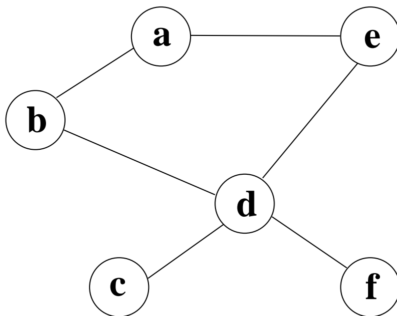
Graphs

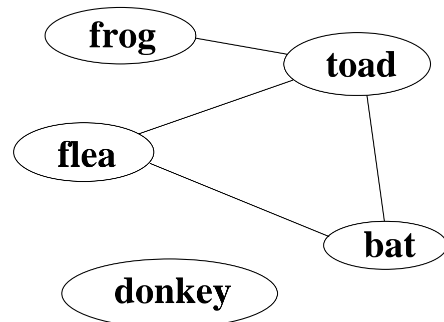
A graph, like a tree, is a nonlinear data structure consisting of nodes and links between the nodes.

23.1 Graph Definition

23.1.1 Undirected Graphs

- A *undirected graph* consists of a set of *nodes* (*vertices* or *points*) and a set of *edges* (*links*) connecting pairs of nodes.
- Example





23.1.2 Directed Graphs

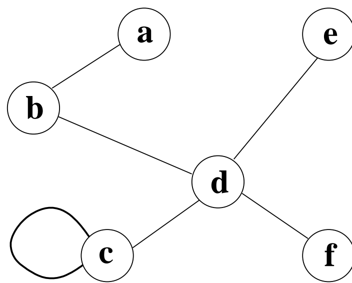
- In an undirected graph each edge connecting two vertices has no particular orientation or direction
- If each edge in a graph has an orientation connecting its first vertex, called the edge's *source*, to its second vertex, called the edge's *destination* or *target*, then the graph is called a *directed graph*.
- Example

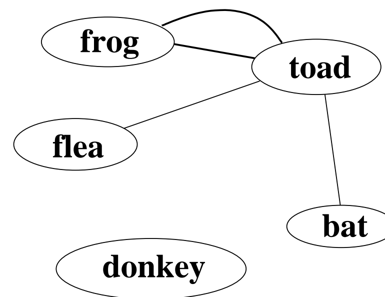
- An undirected graph can be considered as a directed graph if each edge in the undirected graph is considered as two edges (two directions) in the directed graph.

23.1.3 More Terminology

- A *loop* is an edge that connects a vertex to itself.
- A *path* in a graph is a sequence of vertices, v_0, v_1, ..., v_n, such that each consecutive pair of vertices v_i and v_{i+1} are connected by an edge. In a directed graph, the connection must go from the source v_i to the target $v_{i+1}$is a *sequence* of nodes, i.e., (v_i, v_{i+1}) is an edge of the graph
- The *length* of a path is the number of edges in the path.
- In principle, a graph may have two or more edges connecting the same two vertices in the same direction. These are called *multiple edges*.
- If there are no loops and no multiple edges in a (undirected or directed) graph, the graph is called a *simple* graph. We only investigate simple graphs in this course.

must be at most one edge between two nodes





edge must go between two distinct nodes

- Two nodes are *adjacent* if there is an edge between them.

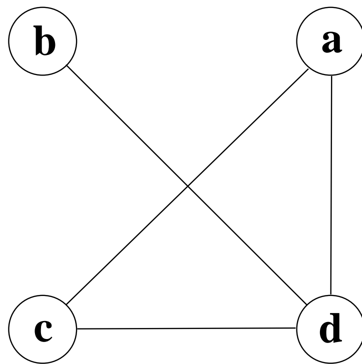
23.2 Graph Implementation

23.2.1 Two Implementations for Graphs

- Static:
  - Uses a 1D array to store the node values
  - Uses a 2D *adjacency matrix* to store the edges
- Dynamic:
  - Uses linked lists

23.2.2 Adjacency Matrices

- An *adjacency matrix* is a 2D boolean array where the ij-element is `TRUE` if and only if there exists an edge between nodes i and j (For a directed graph this means that there exists an edge from the node i to the node j).
- We write
  - `TRUE = 1`
  - `FALSE = 0`
- Example: a Static Graph



count = 4

$$\text{nodes} = \begin{matrix} 0 & 1 & 2 & 3 \\ [a & b & c & d] \end{matrix}$$

$$\text{edges} = \begin{matrix} & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{matrix}$$

You can find another example on page 703.

- The adjacency matrix of an undirected graph is always symmetric. This may be not true for a directed graph. Why?

23.2.3 A Look at the `graph` Class

```
#define NUMNODES 40

template <class Item>
class graph {
  public:
      ...

  private:
      size_t count;
      Item nodes[NUMNODES]
      bool edges[NUMNODES][NUMNODES];
       ...
}
```

23.2.4 Graphs with Dynamic Implementation

- A directed graph with `n` vertices can be represented by `n` different linked lists. List number `i` provides the connections for vertex `i`. Each entry in the linked list `i` is a vertex number `j` such that there is an edge from `i` to `j`.
- The same technique also works with any undirected graphs. We use an array of pointers to the linked lists to represent the graph.
- Following the same idea, a graph can be represented by an array of sets of integers. For example, to represent a graph with 10 vertices, we can declare an array of 10 sets of integers. The `i`-th set contains the vertex numbers of all of the vertices that vertex `i` is connected to.

23.3 Member Functions for a Graph Class

From now on we suppose we use the static implemention of the graph with the adjacency matrix.

23.3.1 Adding Vertices and Edges

- Adding an vertex to an existed graph is much simpler. The work to be done includes updating the `count` by 1, putting the new entry into the `nodes` array at the location `count` and setting all the elements on the `count`-th column and the `count`-th row of the array `edges` to `false`.
- The implementation may look like the following

```
template <class Item>
void graph<Item>::add_vertex(const Item& entry){
   size_t new_count, other_count;
   new_count = count;
   ++count;
   for (other_count=0; other_count < count; ++other_count)
   {
      edges[other_count][new_count] = false;
      edges[new_count][other_count] = false;
   }
   nodes[new_count] = entry;
}
```

- Adding an edge means setting an appropriate element of the array `edges` to the value `true`.

```
template <class Item>
void graph<Item>::add_edge(size_t source, size_t target){
      edges[source][target] = true;
}
```

The above code is for a directed graph. Can you change it for an undirected graph?

- Thus removing an edge from a graph is also very simple. You can simply work it out.
- Other methods include extracting entry information of the `i`-th node and getting all edges for a given node.

Chapter 24

Algorithms for Graphs

24.1 Graph Traversals

24.1.1 Searching for Paths

- At times we may want to find a path between two nodes
- There are two methods:
  - *Depth-first search*: Search as far as possible in each direction
  - *Breadth-first search*: Search nodes closest to the start first

24.1.2 Depth-First Search

Finding a path from vertex v to w

- Process v
- Find an unprocessed node, u, adjacent to v (or the target from v).
- Continue recursively from u as far as possible
- Find the next unprocessed node adjacent to v (or the target from v) and repeat.
- We need to keep track of which vertices have been processed.

24.1.3 Breadth-First Search

Finding a path from vertex v to u

- Process v
- Recursively search each unprocessed node adjacent to v (or the target from v)
- The algorithm continues until the node is found, or we have exhausted every possibility

24.1.4 Implementation

As we discussed in Lecture 23, an undirected graph can be considered as a directed graph. Hence we simply consider the implementation for directed graphs.

- The textbook shows the Depth-First Search algorithm with an example on pages 717—720 and the Breadth-First Search algorithm with another example on pages 720—722.
- The pseudocode for the Depth-First Search is
  1. Check that the start vertex is a valid vertex number of the graph
  2. Set all the components of `marked` array to `false` indicating if a vertex has been processed by the algorithm
  3. Recursively call a separate function to process the neighbors' neighbors until all the vertices are processed.
- The C++ code for the Depth-First Search

```
template <class Process, class Item, class SizeType>
void depth_first(Process f, graph<Item>& g, SizeType start) {
    bool marked[NUMNODES];
    std::fill_n(marked, g.size(), false);
    recur_dfs(f, g, start, marked);
}
template <class Process, class Item, class SizeType>
void recur_dfs(Process f, graph<Item>& g, SizeType v, bool marked[]) {
    set<size_t> connections = g.neighbors(v);
    set<size_t>::iterator it;
    marked[v] = true; //mark vertex v
    f(g[v]);          //process the label of vertex
    for (it=connections.begin(); it != connections.end(); ++it)
    {
        if (!marked[*it])
            recur_dfs(f, g, *it, marked);
    }
}
```

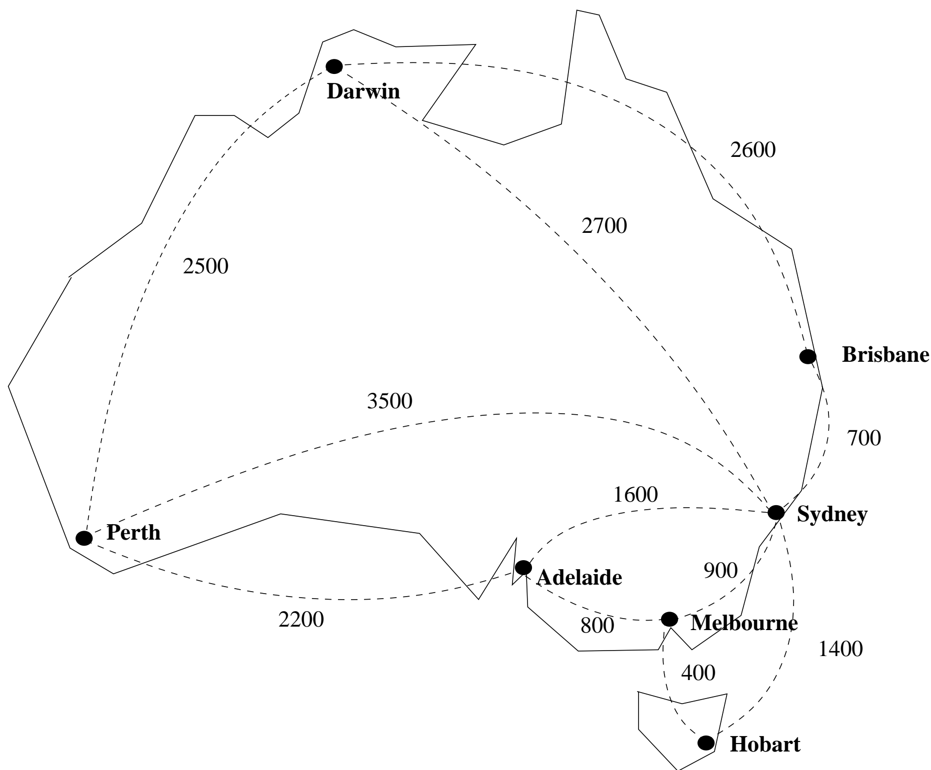
- The Breadth-First Search is implemented with a queue of vertex numbers. The start vertex is processed, marked, and placed in the queue. Then following steps are repeated until the queue is empty
  1. remove a vertex `v` from the front of the queue
  2. for each unmarked neighbor `u` of `v`: process `u`, mark `u` and then place `u` in the queue.
- Here is the code for the Breadth-First Search algorithm

```
template <class Process, class Item, class SizeType>
void breadth_first(Process f, graph<Item>& g, SizeType start) {
     bool marked[NUMNODES];
     set<size_t> connections;
     set<size_t>::iterator it;
     queue<size_t> vertex_queue;
     std::fill_n(marked, g.size(), false);
     marked[start] = true; //mark vertex v
     f(g[start]);
     vertex_queue.push(start);
     do {
        connections = g.neighbors(vertex_queue.front());
        vertex_queue.pop();
        for (it=connections.begin(); it != connections.end(); ++it)
        {
            if (!marked[*it]) {
               marked[*it] = true;
               f(g[*it]);
               vertex_queue.push(*it);
            }
        }
     }  while (!vertex_queue.empty())
}
```

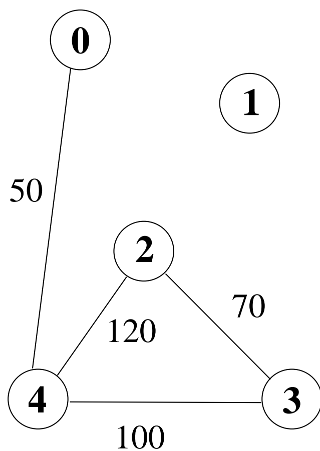
24.2 Path Algorithms

24.2.1 Networks or Weighted Graphs

- A graph having a number associated with each edge is a *network* (aka. *weighted graph*).
- The edge number is called a *weight*.
- Networks can represent many different things:
  - A computer network: Nodes — the computers; Weights — the estimated delay
  - Flight paths: Nodes — the cities; Weights — distance or costs between cities
- Example



- The implementation for a network is similar to graph implementation. The *adjacency matrix* stores numbers instead of booleans.



count = 5

nodes = [0 1 2 3 4]

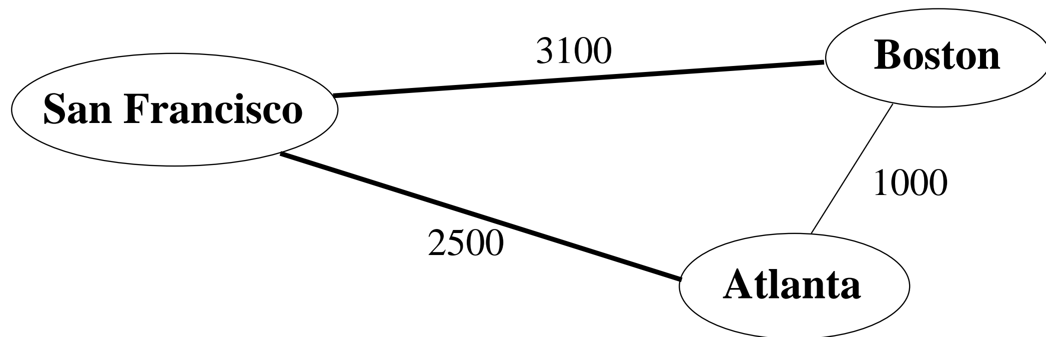
$$\text{edges} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 50 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 70 & 120 \\ 0 & 0 & 70 & 0 & 100 \\ 50 & 0 & 120 & 100 & 0 \end{bmatrix} \end{array}$$

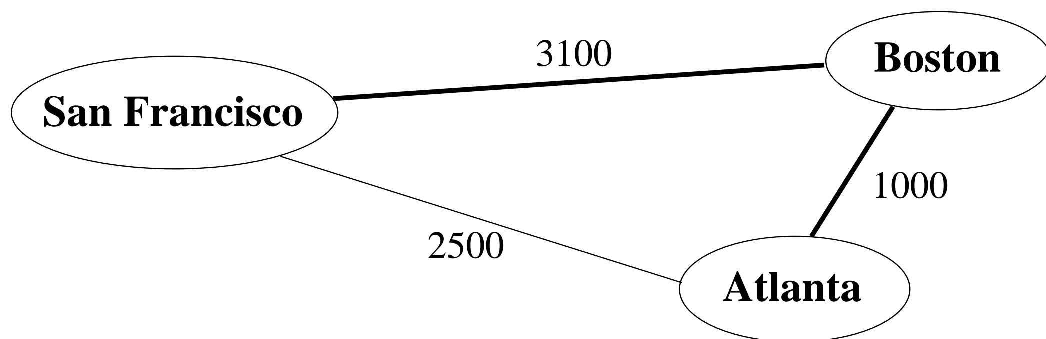
24.2.2 Spanning Trees and Prim's algorithm

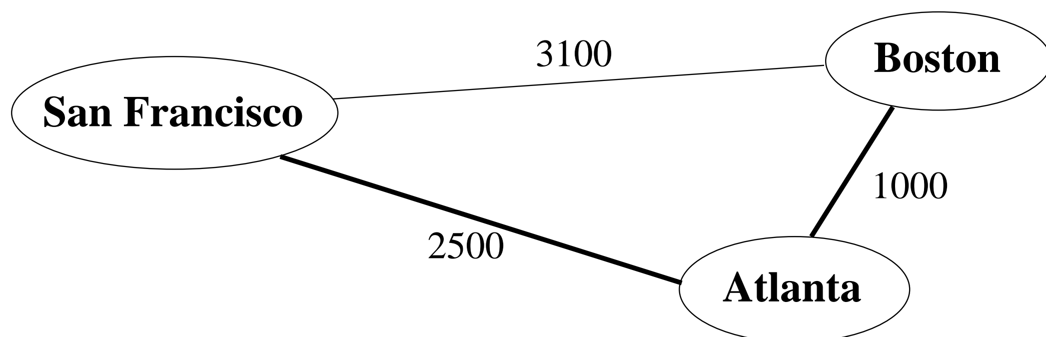
In this subsection we simply considered undirected networks.

- A *tree* is a connected graph with no cycles. A *spanning tree* of a connected graph is a tree whose edges are in the graph and that contains all of the vertices in the graph.
- A *minimal spanning tree* of a connected network is a spanning tree that has the smallest possible weight sum.

- Example: Spanning Trees







- *Prim's algorithm* is used to find a *minimal spanning tree* `t` of a connected network `g` of `n` vertices:

```
find an edge in g with the smallest weight
add it and its two nodes to t
repeat until there are n nodes in t:
  find an edge in g that is incident to exactly one node of
  t and that has the smallest weight
  add it and the new node to t
```

24.2.3 Shortest Path and Dijkstra's Algorithm

- Depth-First Search will find path, but not necessarily shortest path. One may want to find quickest way to target vertex. For example, on a airline network one may want to find a path

between two cities with the most economic cost.

- *Dijkstra's algorithm* finds the *shortest path* from vertex 0 to all other vertices (1, 2, . . ., n-1). A shortest path between two vertices may not cover every vertex of the network because it is notnecessarily to find a spanning tree

- The Pseudocode for the Algorithm

  1. Initialize arrays `distance`, `FromvVertex` and `processed`:

$$distance[i] \leftarrow \begin{cases} 0 & \text{if } i = 0, \\ weight_{0i} & \text{if 0 and } i \text{ are adjacent,} \\ \infty & \text{otherwise} \end{cases}$$

$$FromVertex[i] \leftarrow \begin{cases} 0 & \text{if 0 and } i \text{ are adjacent,} \\ -1 & \text{otherwise} \end{cases}$$

$$processed[i] \leftarrow \begin{cases} TRUE(1) & \text{if } i = 0, \\ FALSE(0) & \text{otherwise} \end{cases}$$

  2. Then carry out the following procedure

```
repeat n-1 times
    IndexMin = index of unprocessed vertex closest to vertex 0
    mark that vertex as having been processed
    for each vertex j from 1 to n-1 do
    if vertex j hasn't been processed AND vertices j &
                 IndexMin  are adj.
       if it's shorter to get to vertex j from 0 via IndexMin
          change distance[j] to that distance
          FromVertex[j] = IndexMin
```

  3. When the algorithm terminates, we follow the `FromVertexs` back to vertex 0.

- The text proposes another version of the algorithm which simply output the shortest distance between start vertex and all other vertices. Please read the pseudocode on page 735 carefully and see the example of Dijkstra's algorithm on pages 729—738.