1 Stack Data Structure

In this section, we first introduce stack data structure and present a template representation for this data structre. We also show some common applications for stack data structure.

1.1 Introduction to Stacks

- Stack is a data structure into which we insert items at the top and retrieve those items in last-in, first-out order independent of the type of the items being placed in the stack.
- To instantiate a stack, a data type must be specified. This creates a wonderful opportunity for software reusability.
- From operation perespective, the following operations are required for stacks
  - **Push operation** adds a new item to the top of the stack, or initializes the stack if it is empty. If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state.
  - **Pop operation** removes an item from the top of the stack. A pop either reveals previously concealed items, or results in an empty stack, but if the stack is empty then it goes into underflow state (It means no items are present in stack to be removed).
  - **Clear operation** removes all stack elements
  - **isEmpty** checks if the stack has data or not.
  - **top** operation (also known as peek) retruns the value of the first element without removing it.
- Stack is very useful in many applications, specifically when data has to be stored and retrived in the reverse order. Example applications include
  - matching delimiter in a program. In C++, delimiters include (),[],{}, and /* .....*/. mismatcing dilimiters indicates a code error. (Example in Drozdek book, ch 4)
  - Adding large numbers (Example in Drozdek book, ch 4)
  - Evaluating mathematical expressions (is explained later.)

1.2 Stack Template Implementation

- In this example, the Stack class-template looks like a conventional class definition, except that it is preceded by the header (line 5)

```
template< typename T >
```

  to specify a class-template definition with type parameter T which acts as a placeholder for the type of the Stack class to be created.

- The type of element to be stored on this Stack is mentioned generically as T throughout the Stack class header and member-function definitions.

```
// Stack class template.
 #ifndef STACK_H
 #define STACK_H

 template< typename T >
 class Stack
 {
 public:
    Stack( int = 10 ); // default constructor (Stack size 10)

    // destructor
    ~Stack()
    {
       delete [] stackPtr; // deallocate internal space for Stack
   } // end ~Stack destructor

    bool push( const T& ); // push an element onto the Stack
    bool pop( T& ); // pop an element off the Stack

    // determine whether Stack is empty
    bool isEmpty() const
    {
       return top == -1;
    } // end function isEmpty

    // determine whether Stack is full
    bool isFull() const
    {
       return top == size - 1;
    } // end function isFull

 private:
    int size; // # of elements in the Stack
    int top; // location of the top element (-1 means empty)
    T *stackPtr; // pointer to internal representation of the Stack
 }; // end class template Stack

 // constructor template
 template< typename T >
 Stack< T >::Stack( int s )
    : size( s > 0 ? s : 10 ), // validate size
      top( -1 ), // Stack initially empty
      stackPtr( new T[ size ] ) // allocate memory for elements
 {
      // empty body
   } // end Stack constructor template

   // push element onto Stack;
   // if successful, return true; otherwise, return false
   template< typename T >
   bool Stack< T >::push( const T &pushValue )
   {
      if ( !isFull() )
      {
         stackPtr[ ++top ] = pushValue; // place item on Stack
         return true; // push successful
      } // end if

      return false; // push unsuccessful
   } // end function template push

   // pop element off Stack;
   // if successful, return true; otherwise, return false
   template< typename T >
   bool Stack< T >::pop( T &popValue )
   {
      if ( !isEmpty() )
      {
         popValue = stackPtr[ top-- ]; // remove item from Stack
         return true; // pop successful
      } // end if

      return false; // pop unsuccessful
   } // end function template pop

   #endif
```

CODING TIPS

A common problem is that a header file is required in multiple other header files that are later included into a source code file, with the result often being that variables, structs, classes or functions appear to be defined multiple times (once for each time the header file is included). This can result in a lot of compile-time headaches. Fortunately, the preprocessor provides an easy technique for ensuring that any given file is included once and only once.

By using the #ifndef directive, you can include a block of text only if a particular expression is undefined; then, within the header file, you can define the expression. This ensures that the code in the #ifndef is included only the first time the file is loaded.

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_

/* code */

#endif // #ifndef _FILE_NAME_H_
```

Notice that it's not necessary to actually give a value to the expression _FILE_NAME_H_. It's sufficient to include the line "#define _FILE_NAME_H_" to make it "defined". (Note that there is an n in #ifndef–it stands for "if not defined").

- Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this Stack
  1. they must have a default constructor (for use in line 43 to create the array that stores the stack elements), and
  2. they must support the assignment operator (lines 54 and 68).
- The member-function definitions of a class template are function templates. The member-function definitions that appear outside the class template definition each begin with the header

```
template< typename T >
```

  (lines 39, 50 and 64). Thus, each definition resembles a conventional function definition, except that the Stack element type always is listed generically as type parameter T.
- When doubleStack is instantiated as type Stack< double >, the Stack constructor function-template specialization uses new to create an array of elements of type double to represent the stack (line 43).

```
stackPtr = new T[ size ];
```

  in the Stack class-template definition is generated by the compiler in the class-template specialization Stack< double > as

```
stackPtr = new double[ size ];
```

- The following code represent a driver for the developed stack template. The driver program begins by instantiating object doubleStack of size 5 (line 10). This object is declared to be of class ***Stack< double > (pronounced "Stack of double")***. The compiler associates type double with type parameter T in the class template to produce the source code for a Stack class of type double. ***Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.***

```
 1 // Stack class template test program.
 2 #include <iostream>
 3 using std::cout;
 4 using std::endl;
 5
 6 #include "Stack.h"  // Stack class template definition
 7
 8 int main()
 9 {
10    Stack< double > doubleStack( 5 ); // size 5
11    double doubleValue = 1.1;
12
13    cout << "Pushing elements onto doubleStack\n";
14
15    // push 5 doubles onto doubleStack
16    while ( doubleStack.push( doubleValue ) )
17    {
18       cout << doubleValue << ' ';
19       doubleValue += 1.1;
20    } // end while
21
22    cout << "\nStack is full. Cannot push " << doubleValue
23       << "\n\nPopping elements from doubleStack\n";
24
25    // pop elements from doubleStack
26    while ( doubleStack.pop( doubleValue ) )
27       cout << doubleValue << ' ';
28
29    cout << "\nStack is empty. Cannot pop\n";
30
31    Stack< int > intStack; // default size 10
32    int intValue = 1;
33    cout << "\nPushing elements onto intStack\n";
34
35    // push 10 integers onto intStack
36    while ( intStack.push( intValue ) )
37    {
38       cout << intValue << ' ';
39       intValue++;
```

```
    } // end while

    cout << "\nStack is full. Cannot push " << intValue
       << "\n\nPopping elements from intStack\n";

    // pop elements from intStack
    while ( intStack.pop( intValue ) )
       cout << intValue << ' ';

    cout << "\nStack is empty. Cannot pop" << endl;
    return 0;
} // end main
```

1.3 Other Stack Implememtations

- Linked list implementation (Discussion)

1.4 Evaluating Mathematical expressions using stacks

1.4.1 Introduction

Mathematical expression can be written in different notations including Infix, Postfix and Prefix. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

- **Infix notation: X + Y**
  - Operators are written in-between their operands.
  - This is the usual way we write expressions.
  - An expression such as A * (B + C) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."
  - Infix notation needs extra information to make the order of evaluation of the operators clear: rules built into the language about operator precedence and associativity, and brackets ( ) to allow users to override these rules.
  - For example, the usual rules for associativity say that we perform operations from left to right, so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.
- **Postfix notation (also known as "Reverse Polish notation (RPN)": X Y +**
  - Operators are written after their operands.
  - The infix expression given above is equivalent to A B C + * D /

   - The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.
   - Operators act on values ***immediately*** to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit: ((A (B C +) *) D /)
   - Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

- **Prefix notation (also known as "Polish notation"): + X Y**
   - Operators are written before their operands. The expressions given above are equivalent to / * A + B C D
   - As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear: (/ (* A (+ B C)) D)
   - Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).
   - Because Postfix operators use values to their left, any values involving computations will already have been calculated as we go left-to-right, and so the order of evaluation of the operators is not disrupted in the same way as in Prefix expressions.

- *Conversion from one form to another may be accomplished using a stack.*
  In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - B does not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A).

1.4.2 Evaluation of an infix expression that is fully parenthesized

Analysis: Five types of input characters

Opening bracket

Numbers

Operators

Closing bracket

New line character [\n]

Data structure requirement: A character stack

Algorithm 1 Evaluation of an infix expression that is fully parenthesized

```
1. Read one input character
2. Actions at end of each input
    (a) Opening brackets (2.a) Push into stack and then Go to step (1)
    (b) Number (2.b) Push into stack and then Go to step (1)
    (c) Operator (2.c) Push into stack and then Go to step (1)
    (d) Closing brackets, Pop is used four times
        The first popped element is assigned to op2
        The second popped element is assigned to op
        The third popped element is assigned to op1
        The fourth popped element is the remaining opening bracket, which can be discarded
        Evaluate op1 op op2
        push the result into the stack
        Go to step 1
    (e) New line character (2.e) Pop from stack and print the answer and STOP
```

Eample

Input: (((2 * 5) - (1 * 2)) / (11 - 9))
Output: 4

Input Symbol	Stack (from bottom to top)	Operation
(	(	
(	((	
(	(((	
2	(((2	
*	(((2 *	
5	(((2 * 5	
)	((10	2 * 5 = 10 and push
-	((10 -	
(	((10 - (	
1	((10 - (1	
*	((10 - (1 *	
2	((10 - (1 * 2	
)	((10 - 2	1 * 2 = 2 & Push
)	(8	10 - 2 = 8 & Push
/	(8 /	
(	(8 / (	
11	(8 / (11	
-	(8 / (11 -	
9	(8 / (11 - 9	
)	(8 / 2	11 - 9 = 2 & Push
)	4	8 / 2 = 4 & Push
New line	Empty	Pop & Print

1.4.3 Evaluation of infix expression which is not fully parenthesized

Input: (2 * 5 - 1 * 2) / (11 - 9)

Output: 4

Analysis: There are five types of input characters which are:

Opening brackets

Numbers

Operators

Closing brackets

New line character (\n)

- By implementing the priority rule for operators, we can evaluate the problem. To do so, we evaluate the expressions using two stacks, an integer stack that holds the operand and a character stack that holds the operations and parentheses.
- The Priority rule: we should perform a comparative priority check if an operator is read, and then push it.

Algorithm 2 Evaluation of infix expression which is not fully parenthesized

```
1. Read an input character
2. Actions that will be performed at the end of each input
   (a) Opening brackets (2.a) Push it into character stack and then Go to step (1)
   (b) Number (2.b) Push into integer stack, Go to step (1)
   (c) Operator (2.c) Do the comparative priority check
       i. if the character stack's top contains an operator with equal or higher priority, then
          pop it into op
       ii. Pop a number from integer stack into op2
       iii. Pop another number from integer stack into op1
       iv. Calculate op1 op op2 and push the result into the integer stack
       v. push the operator read in (2.c) to the character stack
   (d) Closing brackets (2.d) Pop from the character stack
       i. (2.d.i ) if it is an opening bracket, then discard it and Go to step (1)
       ii. (2.d.ii) assign the popped element to op
       iii. Pop a number from integer stack and assign it op2
       iv. Pop another number from integer stack and assign it to op1
       v. Calculate op1 op op2 and push the result into the integer stack
       vi. Go to the step (2.d)
   (e) /n (2.5) Print the result after popping from the stack and STOP
```

Input	**Chr Stack**	**Int Stack**	**Operation performed**
(	**(**		
2	**(**	**2**	
*****	**(***		**Push ***
5	**(***	**2 5**	
-	**(***		**Since '-' has less priority, we do 2 * 5 = 10**
	(-	**10**	**We push 10 and then push '-'**
1	**(-**	**10 1**	
*****	**(- ***	**10 1**	**Push * as it has higher priority**
2	**(- ***	**10 1 2**	
)	**(-**	**10 2**	**Perform 1 * 2 = 2 and push it**
	(	**8**	**Pop - and 10 - 2 = 8 and push, Pop (**
/	**/**	**8**	
(	**/ (**	**8**	
11	**/ (**	**8 11**	
-	**/ (-**	**8 11**	
9	**/ (-**	**8 11 9**	
)	**/**	**8 2**	**Perform 11 - 9 = 2 and push it**
/n		**4**	**Perform 8 / 2 = 4 and push it**
		4	**Print the output, which is 4**

1.4.4 Evaluation of prefix expression

Example: / - * 2 5 * 1 2 - 11 9

Algorithm 3 Evaluation of prefix expression

```
1. Read one character input at a time and keep pushing it into the character stack until the
   new line character is reached
2. Perform pop from the character stack. If the stack is empty, go to step (3)
   (a) Number (2.1) Push in to the integer stack and then go to step (2)
   (b) Operator (2.2) Assign the operator to op
       i. Pop a number from integer stack and assign it to op1
       ii. Pop another number from integer stack and assign it to op2
       iii. Calculate op1 op op2 and push the output into the integer stack. Go to step (2)
3. Pop the result from the integer stack and display the result
```

The following table shows the execution after reading the expression to the character stack (end of step 1)

Chr stack	**Int Stack**	**Operation**
/ - * 2 5 * 1 2 - 11 9	**9**	
/ - * 2 5 * 1 2 -	**9,11**	
/ - * 2 5 * 1 2	**2**	**11-9=2**
/ - * 2 5* 1	**2,2**	
/ - * 2 5*	**2,2,1**	
/ - * 2 5	**2,2**	**1*2=2**
/ - * 2	**2,2,5**	
/ - *	**2,2,5,2**	
/ -	**2,2,10**	**5*2=10**
/	**2,8**	**10-2=8**
	4	**8/2=4**
		print 4

1.4.5 Evaluation of postfix expression

Algorithm 4 Evaluation of postfix expression

```
1. when encountering an operand: push it
2. when encountering an operator: pop two operands, evaluate the result and push it.
```

1.4.6 Conversion of an Infix expression that is fully parenthesized into a Postfix expression

Algorithm 5 Infix to postfix conversion

```
1. Read a character input
2. Actions to be performed at end of each input
    (a) Opening brackets (2.a) Push into stack and then Go to step (1)
    (b) Number (2.b) Print and then Go to step (1)
    (c) Operator (2.c) Push into stack and then Go to step (1)
    (d) Closing brackets (2.d) Pop a character from the stack
        i. (2.d.i) If it is an operator, print it, Go to step (2.d)
        ii. (2.d.ii) If the popped element is an opening bracket, discard it and go to step (1)
    (e) New line character (2.5) STOP
3.
```

Example

- Input: (((8 + 1) - (7 - 4)) / (11 - 9))

- Output: 8 1 + 7 4 - - 11 9 - /

Input	Operation	Stack (after op)	Output on monitor
(	(2.1) Push operand into stack	(	
(	(2.1) Push operand into stack	((	
(	(2.1) Push operand into stack	(((	
8	(2.2) Print it		8
+	(2.3) Push operator into stack	(((+	8
1	(2.2) Print it		8 1
)	(2.4) Pop from the stack: Since popped element is '+' print it	(((	8 1 +
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((	8 1 +
-	(2.3) Push operator into stack	((-	
(	(2.1) Push operand into stack	((- (	
7	(2.2) Print it		8 1 + 7
-	(2.3) Push the operator in the stack	((- (-	
4	(2.2) Print it		8 1 + 7 4
)	(2.4) Pop from the stack: Since popped element is '-' print it	((- (	8 1 + 7 4 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((-	
)	(2.4) Pop from the stack: Since popped element is '-' print it	((	8 1 + 7 4 - -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(	
/	(2.3) Push the operand into the stack	(/	
(	(2.1) Push into the stack	(/ (	
11	(2.2) Print it		8 1 + 7 4 - - 11
-	(2.3) Push the operand into the stack	(/ (-	
9	(2.2) Print it		8 1 + 7 4 - - 11 9
)	(2.4) Pop from the stack: Since popped element is '-' print it	(/ (	8 1 + 7 4 - - 11 9 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(/	
)	(2.4) Pop from the stack: Since popped element is '/' print it	(	8 1 + 7 4 - - 11 9 - /
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	Stack is empty	
/n	(2.5) STOP		

2 Queues

2.1 Introduction

- Queues appears naturally in many practical contexts such as person queues, job queues, packet queues, message queues, and many similar contexts.
- A queue is a data structure in which new elements are added to the rear and are removed from the front.
- The queue operates as a FIFO (First Input First Output) system
- The main queue operations are
  - enqueue(element) adds element to the rear of the queue
  - dequeue() removes an element from the front of the queue
  - isEmpty() determines if the queue is empty
  - isFull() determines if the queue reaches its maximum number of elements

2.2 Queue Linked List Implementation

Only the header of the base list template file is shown here, the full template list implementation will be posted to the course website

- Listnode implementation

```
 1  // Template ListNode class definition.
 2  #ifndef LISTNODE_H
 3  #define LISTNODE_H
 4   // forward declaration of class List required to announce that class
 5  // List exists so it can be used in the friend declaration at line 13
 6      template< typename NODETYPE > class List;
 7
 8  template< typename NODETYPE>
 9  class ListNode
10  {
11     friend class List< NODETYPE >; // make List a friend
12
13  public:
14     ListNode( const NODETYPE & ); // constructor
15     NODETYPE getData() const; // return data in node
16  private:
17     NODETYPE data; // data
18     ListNode< NODETYPE > *nextPtr; // next node in list
19  }; // end class ListNode
20
21  // constructor
22  template< typename NODETYPE>
```

```
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 )
{
   // empty body
} // end ListNode constructor
// return copy of data in node
template< typename NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const
{
   return data;
} // end function getData

#endif
```

- Template List header

```
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>
using std::cout;

#include "listnode.h" // ListNode class definition

template< typename NODETYPE >
class List
{
public:
   List(); // constructor
   ~List(); // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
private:
   ListNode< NODETYPE > *firstPtr; // pointer to first node
   ListNode< NODETYPE > *lastPtr; // pointer to last node

   // utility function to allocate new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
}; // end class List

```

```

#endif
```

1 Queues

1.1 Introduction

- Queues appears naturally in many practical contexts such as person queues, job queues, packet queues, message queues, and many similar contexts.
- A queue is a data structure in which new elements are added to the rear and are removed from the front.
- The queue operates as a FIFO (First Input First Output) system
- The main queue operations are
  - enqueue(element) adds element to the rear of the queue
  - dequeue() removes an element from the front of the queue
  - isEmpty() determines if the queue is empty
  - isFull() determines if the queue reaches its maximum number of elements

1.2 Queue Linked List Implementation

Only the header of the base list template file is shown here, the full template list implementation will be posted to the course website

- Listnode implementation

```
// Template ListNode class definition.
#ifndef LISTNODE_H
#define LISTNODE_H
 // forward declaration of class List required to announce that class
// List exists so it can be used in the friend declaration at line 13
    template< typename NODETYPE > class List;

template< typename NODETYPE>
class ListNode
{
   friend class List< NODETYPE >; // make List a friend

public:
   ListNode( const NODETYPE & ); // constructor
   NODETYPE getData() const; // return data in node
private:
   NODETYPE data; // data
   ListNode< NODETYPE > *nextPtr; // next node in list
}; // end class ListNode

// constructor
template< typename NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
   : data( info ), nextPtr( 0 )
{
   // empty body
} // end ListNode constructor
// return copy of data in node
template< typename NODETYPE >
NODETYPE ListNode< NODETYPE >::getData() const
{
   return data;
} // end function getData
```

```
#endif
```

- Template List header

```
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>
using std::cout;

#include "listnode.h" // ListNode class definition

template< typename NODETYPE >
class List
{
public:
   List(); // constructor
   ~List(); // destructor
   void insertAtFront( const NODETYPE & );
   void insertAtBack( const NODETYPE & );
   bool removeFromFront( NODETYPE & );
   bool removeFromBack( NODETYPE & );
   bool isEmpty() const;
   void print() const;
private:
   ListNode< NODETYPE > *firstPtr; // pointer to first node
   ListNode< NODETYPE > *lastPtr; // pointer to last node

   // utility function to allocate new node
   ListNode< NODETYPE > *getNewNode( const NODETYPE & );
}; // end class List


#endif
```

- Queue Implenetation

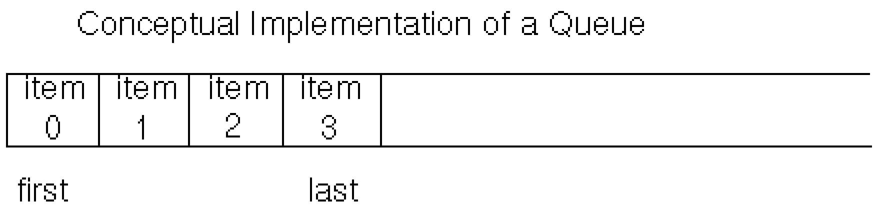
```
#ifndef QUEUE_H
  #define QUEUE_H

  #include "List.h" // List class definition

  template< typename QUEUETYPE >
  class Queue : private List< QUEUETYPE >
  {
  public:
     // enqueue calls List member function insertAtBack
     void enqueue( const QUEUETYPE &data )
     {
        insertAtBack( data );
     } // end function enqueue

     // dequeue calls List member function removeFromFront
     bool dequeue( QUEUETYPE &data )
```

```
   {
      return removeFromFront( data );
   } // end function dequeue

   // isQueueEmpty calls List member function isEmpty
   bool isQueueEmpty() const
   {
      return isEmpty();
   } // end function isQueueEmpty

   // printQueue calls List member function print
   void printQueue() const
   {
      print();
   } // end function printQueue
}; // end class Queue

#endif
```

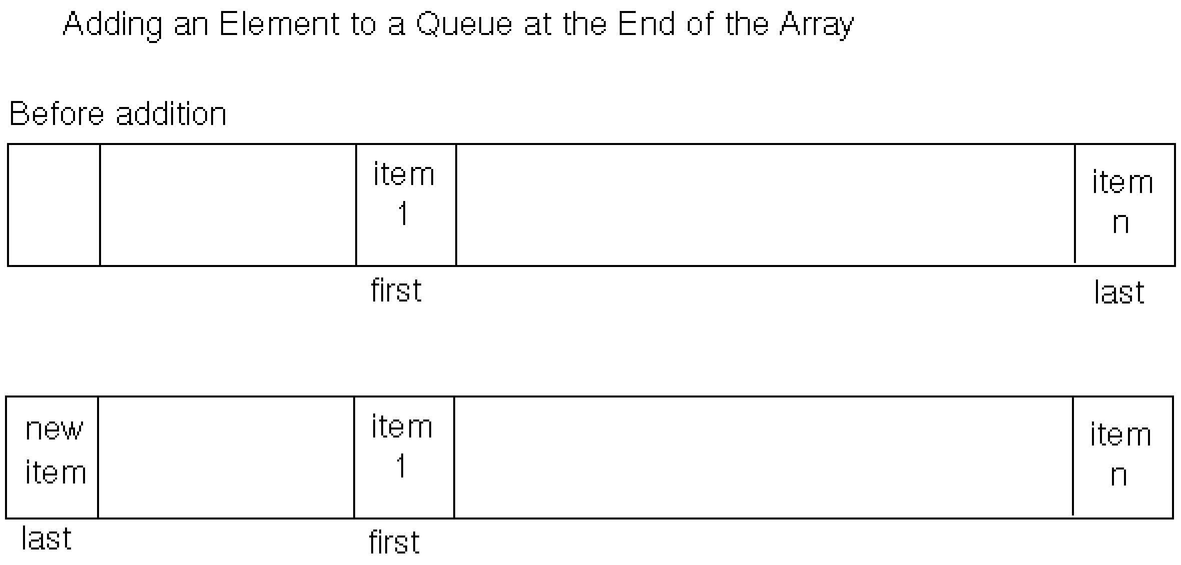
1.3 Queue Array Implementation

The array implementation of queue is discussed in different references. The main ideas in such implementation include

- defining a front and rear indecies to manage enqueue and dequeue



- managing wraping around on reaching the end of the queue (circular queue)



- how to determine the queue is full or empty?

1.4 Priority Queues

- Priority queues are a special type of queues in which queue elements are processed in order of importance/priority
- The priority queues appears in different contexts
  - packets with different priority
  - patients at emergency section
- Implementation Approaches
  - Unsorted list
    - Adv: simple insert
    - Disadv: search before dequeue
  - Linked sorted list
    - Adv: simple dequeue (always get the first element)
    - Disadv: $O(N)$ enqueue as we need to decide where to insert the received object

Appendix- Freindship and inheritance

Friends

- Friends are functions or classes declared with the friend keyword.
- Friends of class A have access to the protected and private members of class A.
- Friendships are not transitive: The friend of a friend is not considered to be a friend unless explicitly specified.

Inheritance

- Inheritance is a mechanism of reusing and extending existing classes without modifying them.
- Inheritance is almost like embedding an object into a class. Suppose that you declare an object x of class A in the class definition of B. As a result, class B will have access to all the public data members and member functions of class A. However, in class B, you have to access the data members and member functions of class A through object x.

```
#include <iostream>
using namespace std;

class A {
   int data;
public:
   void f(int arg) { data = arg; }
   int g() { return data; }
};

class B {
public:
   A x;
};

int main() {
   B obj;
   obj.x.f(20);
   cout << obj.x.g() << endl;
//    cout << obj.g() << endl;
}
```

- inheritance mechanism lets you use a statement like obj.g() in the above example. In order for that statement to be legal, g() must be a member function of class B

  ```
  #include <iostream>
  using namespace std;

  class A {
     int data;
  public:
     void f(int arg) { data = arg; }
     int g() { return data; }
  };

  class B : public A { };

  int main() {
     B obj;
     obj.f(20);
     cout << obj.g() << endl;
  }
  ```

  - Class A is a ***base class*** of class B. The names and definitions of the members of class A are included in the definition of class B;
  - class B inherits the members of class A.
  - Class B is derived from class A.
  - Class B contains a subobject of type A.
  - You can also add new data members and member functions to the derived class.
  - You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

- a derived class inherits every member of a base class **except**
  - its constructor and its destructor.
    * However, the default constructor (i.e., its constructor with no parameters) and destructor of the base class are always called when a new object of a derived class is created or destroyed.
    * The base default constructor can be overridden.
  - its operator=() members
  - its friends

References

[1] Nell Dale "C++ Plus Data Structures," 3rd Edition (2003)

[2] Adam Drozdek, "Data Structures and Algorithms in C++," 2nd Edition

Search Algorithms and their Complexities

May 13, 2013

1 Algorithm Complexity

- Informally, an algorithm can be said to exhibit a growth rate on the order of a mathematical function if beyond a certain input size n, the function f(n) times a positive constant provides an upper bound or limit for the run-time of that algorithm. In other words, for a given input size n greater than some n0 and a constant c, the running time of that algorithm will never be larger than c × f(n). This concept is frequently expressed using ***Big O*** notation.
- In computer science, big-O notation is used to classify algorithms by how they respond (i.e., in their processing time or working space requirements) to changes in input size.

1.1 Calculating the algorithm complexity

- Each instruction has well defined execution time. However, the specific amount of time to carry out a given instruction will vary depending on which instruction is being executed and which computer is executing it.
- To determine the time complexity of an algorithm:
  - Express the amount of work done as a sum $f1(n) + f2(n) + \ldots + fk(n)$
  - Identify the dominant term f_j whose complexity is $O(f_j)$ and satisfies

$$f_k(n) < f_j(n)\ \forall k$$

  - Then the time complexity is $O(f_j)$
- Consider the following pseudocode

```
1     get a positive integer from input
2     if n > 10
3         print "This might take a while..."
4     for i = 1 to n
5         for j = 1 to i
6             print i * j
7     print "Done!"
```

  - instructions 1-3 and 7 have fixed execution time independent of the problem size n
  - outer loop executes $(n+1)$ times (note the last check)
  - the inner loop executes Instruction 6 i-times for every time the outer loop is visited
  - hence, the total running time of line 6 can be calculated as

$$T_6[1+2+...+n] = T_6\frac{(n^2+n)}{2}$$

- Big-O Analysis in General
  - find the section of code that you expect to have the highest order. From there, work out the algorithmic efficiency from the outside in
  - figure out the complexity of the outer loop or recursive portion of the code, then
  - find the complexity of the inner code;
  - the total complexity is the complexity of each layer of code multiplied together.

```
int x = 0;
for ( int j = 1; j <= n/2; j++ )
for ( int k = 1; k <= n*n; k++ )
x = x + j + k;
```

Outer loop executes $n/2$ times. For each of those times, inner loop executes n^2times, so the body of the inner loop is executed $(n/2) * n^2 = n^3/2$ times. The algorithm is $O(n^3)$.

1.2 Common complxity levels are

- some functions perform independent of the input size. Such functions are said to have $\Theta(1)$ complixity and are known as bounded time functions.
- $O(log_2N)$ is also another common complexity level that commonly appears in the operations that can split the data into halves and proceed with half of the data. Such algorithms are said to have a *logarithmic compplexity.*
- $O(N)$ operations are said to have *linear complexity* and typically every data element is visited once. Printing a list element is an example for such operations.
- $O(Nlog_2N)$ is also another common complexity level that is very common with efficient sorting algorithms as will be studied later.
- $O(N^2)$ is a polynomial complexity if the second order.
- $O(2^N)$ represents exponential complexity.

1.3 ADT Operation complexity

- Let's consider the queue ADT for example, the required operations are enqueue, dequeu, isFull, isEmpty, clear, constructor and destructor
  - The complexity of all the operations is $O(1)$ independent of the implementation except for the clear and the destructor operation of the linked implementation. Note that in the latter functions, the nodes has to be traversed one by one resulting in a complexity of $O(N)$ for these operations.
- What about queue array implementation?
- What about queue circular array implementation?

2 Searching Algorithms

- Searching data involves determining whether a value (referred to as the ***search key***) is present in the data and, if so, finding the value's location
- Two popular search algorithms are the simple linear search and the faster but more complex binary search.

2.1 Linear Search

- The linear search compares each element of an array with a search key.
- Linear Search is typically used if the array is not in any particular order. Therefore, the program must compare the search key with half the elements of the array. To determine that a value is not in the array, the program must compare the search key to every element in the array.
- Clearly, the linear search is of complexity $O(N)$

2.2 Binary Search

- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the vector first be sorted.
  - The first iteration of this algorithm tests the middle element in the vector.
  - If this matches the search key, the algorithm ends.
  - Assuming the vector is sorted in ascending order, then if the search key is less than the middle element, the search key cannot match any element in the second half of the vector and the algorithm continues with only the first half of the vector (i.e., the first element up to, but not including, the middle element).
  - If the search key is greater than the middle element, the search key cannot match any element in the first half of the vector and the algorithm continues with only the second half of the vector (i.e., the element after the middle element through the last element).

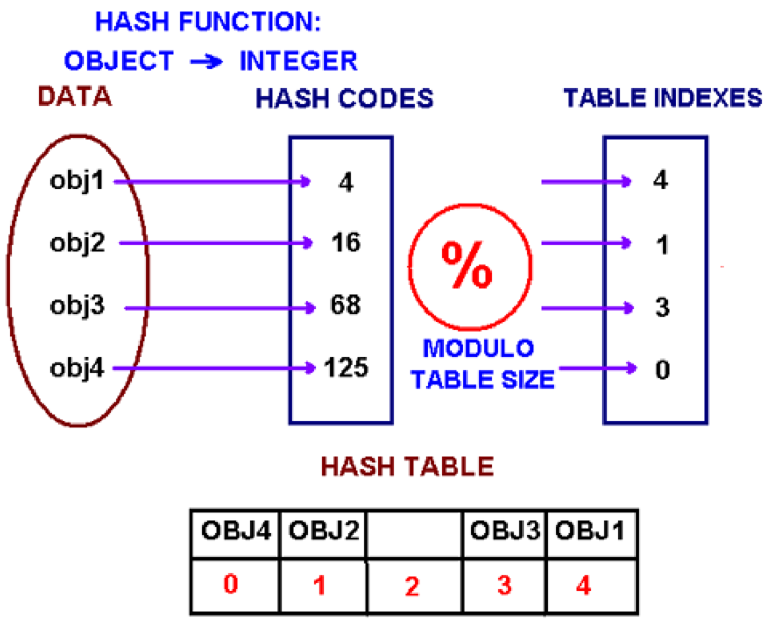
  - Each iteration tests the middle value of the remaining portion of the vector.

- Example: Considering the sorted array [4 5 10 30 34 51 52 56 77 93]. On searching 30, the item will be located in ...... 4 ...... iterations using binary search.
- The complexity of binary search is $O(log_2 N)$.

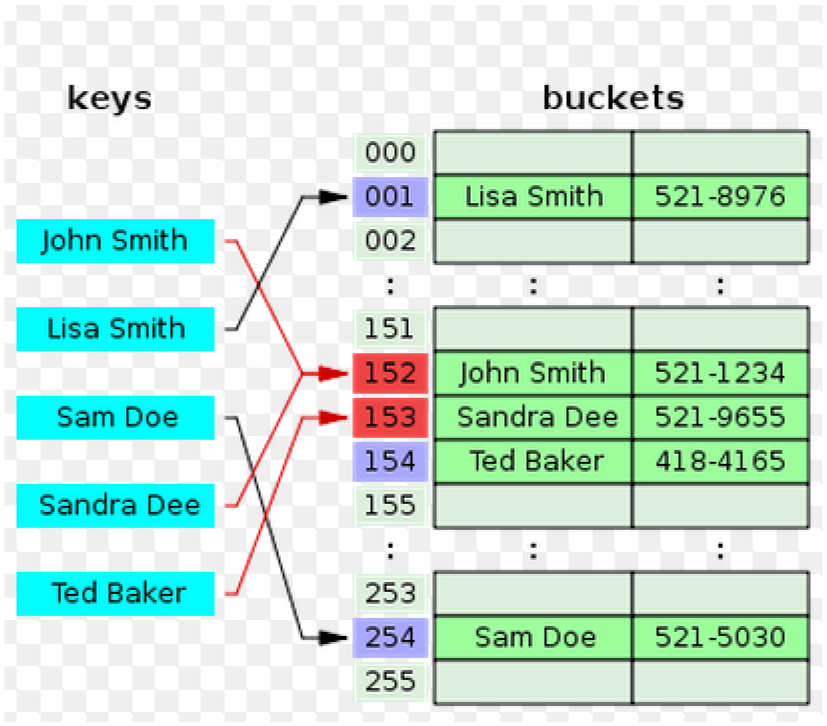
- Selecting Element to compare with
  - If the list/array has odd number of elements, the comparison is made with the middle element
  - if the list/array has even number of ellements, you compare with the last element in the first array
- After comparison, the nonmatching element is ignored and the selection is made from the elements above or below the compared element. For example, on comparing 30 and 34 in the first trial in the example above, the list of considered elements in the next trial is [4 5 10 30] excluding 34.

3 Hashing

- Hashing is a technique that enables searching the data with a complexity $O(1)$
- Using a key (index) value for each a data record, the key value can be used to represent an array index that is used for data insertion (no sorting time) and retrieval (no search time)
- The challenge in this design is to define a ***mapping function*** between the steored record data and the key/index value. Such a mapping function is called ***hash function.***
- For illustration, let's assume that we need to define a hash function for the student records of a class consisting of 100 students. One possible design is to map the student number (which generally consists of more digits, e.g. 987635). A typical hash function can expressed as $hash(x) = x\%$.



- However, the proposed has function may result in the same hash value for two different records. This situation is known as ***collision***.
- Generally, a collision-free hash finction is not easy to design.
  - a usual approach is to use an array size that is larger than needed. The extra array positions make the collisions less likely
  - A good hash function will distribute the keys uniformly throughout the locations of the array
- **Collison resolution.** One way to resolve collisions is to place the colliding record in another location that is still open. This storage algorithm is called ***open-addressing.***
  - Open addressing requires that the array be initialized so that the program can test if an array position already contains a record.
  - A mechanism is required to determine the next array index to be searched.
    - **linear probing**. if $hash(x)$ is full check for $[hash(x) + 1]$.
      A common problem with linear probing is clustering by which the colliding elements forms clusters rather than being evenly distributed in the array. Clustering makes insertions take longer because the insert function must step all the way through a cluster to find a vacant location. Searches require more time for the same reason.



  * **double hashing**. double hashing is the most common technique to avoid clustering. Let $i = hash(key)$. If the location $data[i]$ already contains a record then let $i = (i + hash2(key))\%$, and try the new $data[i]$. If that location already contains a record, then let $i = (i + hash2(key))\%$, and try that $data[i]$, and so forth until a vacant position is found.

- Chained hashing
  - In open addressing, each array element can hold just one entry. When the array is full, no more records can be added to the table.
  - One possible solution is to resize the array and rehash all the entries. This would require a careful choice of new size and probably require each entry to have a new hash value computed.
  - A better approach is to use a different collision resolution method called chained hashing, or simply chaining, in which each component of the hash table's array can hold more than one entry. We still hash the key of each entry, but upon collision, we simply place the new entry in its proper array component along with other entries that happened to hash to the same array index.
  - The most common way to implement chaining is to have each array element be a linked list. The nodes in a particular linked list will each have a key that hashes to the same value.

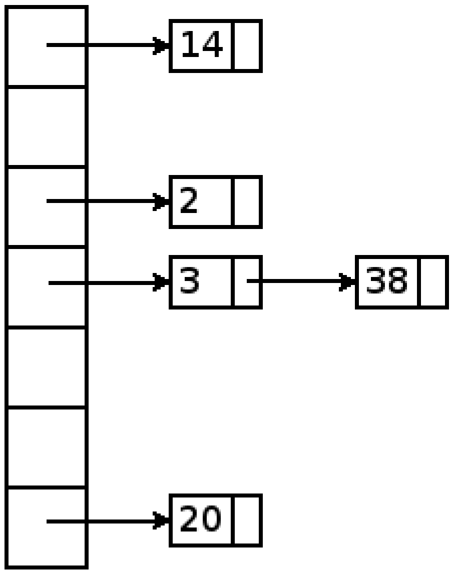


Figure 1: Chained Hashing

```
class LinkedHashEntry {
private:
      int key;
      int value;
      LinkedHashEntry *next;
public:
      LinkedHashEntry(int key, int value) {
            this->key = key;
            this->value = value;
            this->next = NULL;
      }

      int getKey() {
            return key;
      }

      int getValue() {
            return value;
      }

      void setValue(int value) {
            this->value = value;
      }

      LinkedHashEntry *getNext() {
            return next;
      }

      void setNext(LinkedHashEntry *next) {
        this->next = next;
    }
};
```

```
const int TABLE_SIZE = 128;

class HashMap {
private:
    LinkedHashEntry **table;
public:
    HashMap() {
        table = new LinkedHashEntry*[TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i++)
            table[i] = NULL;
    }

    int get(int key) {
        int hash = (key % TABLE_SIZE);
        if (table[hash] == NULL)
            return -1;
        else {
            LinkedHashEntry *entry = table[hash];
            while (entry != NULL && entry->getKey() != key)
                entry = entry->getNext();
            if (entry == NULL)
                return -1;
            else
                return entry->getValue();
        }
    }

    void put(int key, int value) {
        int hash = (key % TABLE_SIZE);
        if (table[hash] == NULL)
            table[hash] = new LinkedHashEntry(key, value);
        else {
            LinkedHashEntry *entry = table[hash];
            while (entry->getNext() != NULL)
                entry = entry->getNext();
            if (entry->getKey() == key)
                entry->setValue(value);
            else
                entry->setNext(new LinkedHashEntry(key, value))
                    ;
        }
    }
```

```

      void remove(int key) {
            int hash = (key % TABLE_SIZE);
            if (table[hash] != NULL) {
                  LinkedHashEntry *prevEntry = NULL;
                  LinkedHashEntry *entry = table[hash];
                  while (entry->getNext() != NULL && entry->getKey() !=
                      key) {
                        prevEntry = entry;
                        entry = entry->getNext();
                  }
                  if (entry->getKey() == key) {
                        if (prevEntry == NULL) {
                             LinkedHashEntry *nextEntry = entry->
                                getNext();
                             delete entry;
                             table[hash] = nextEntry;
                        } else {
                             LinkedHashEntry *next = entry->getNext();
                              delete entry;
                             prevEntry->setNext(next);
                        }
                  }
            }
      }

      ~HashMap() {
            for (int i = 0; i < TABLE_SIZE; i++)
                  if (table[i] != NULL) {
                        LinkedHashEntry *prevEntry = NULL;
                        LinkedHashEntry *entry = table[i];
                        while (entry != NULL) {
                             prevEntry = entry;
                             entry = entry->getNext();
                             delete prevEntry;
                        }
                  }
            delete[] table;
      }
};
```

References

[1] Hashing tutorial, http://research.cs.vt.edu/AVresearch/hashing/

1 Sorting Algorithms

- ***Sorting*** places data in order, typically ascending or descending, based on one or more ***sort keys***.
- Typically, every organization must sort some data, and often, massive amounts of it.
- Sorting is required as it typically reduces search time.
- Sorting large amount of data mat be time consuming. Hence, a fast and efficient sorting algorithms is always desirable
- There exists many sorting algorithms including *insertion sort, selection sort, bubble sort, binary tree sort, and quick sort.*
- On comparing sorting algorithms, one consider a list of N elements and estimate different metrics including the number of comparisons made or the number of swap operation performed.
- The efficieny is may also consider the memory utilization/requirement as a metric on evaluating sorting algorithms

1.1 Selection Sort

- Selection sort algorithm, which is an easy-to-program, but unfortunately inefficient, sorting algorithm.
- The algorithm iterates over an array of elements as follows
  - The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element.
  - The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element.
  - The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.
  - After the ith iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.

1.1.1 Example: Sort 34 56 4 10 77 51 93 30 5 52

1. **4** 56 **34** 10 77 51 93 30 5 52
   (a) 4 **5** 34 10 77 51 93 30 **56** 52
   (b) 4 5 **10** **34** 77 51 93 30 56 52
   (c)
   (d)
   (e)
   (f) 4 5 10 30 34 51 52 56 77 93

1.1.2 Selection Sort Implementation

```
  // selectionSort.cpp
  // This program puts values into an array, sorts the values into
  // ascending order and prints the resulting array.
  #include <iostream>
  using std::cout;
  using std::endl;

  #include <iomanip>
  using std::setw;

  void selectionSort( int * const, const int ); // prototype
  void swap( int * const, int * const ); // prototype

  int main()
  {
     const int arraySize = 10;
     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

     cout << "Data items in original order\n";

     for ( int i = 0; i < arraySize; i++ )
        cout << setw( 4 ) << a[ i ];

     selectionSort( a, arraySize ); // sort the array

     cout << "\nData items in ascending order\n";

     for ( int j = 0; j < arraySize; j++ )
        cout << setw( 4 ) << a[ j ];

     cout << endl;
     system("PAUSE");
    return EXIT_SUCCESS;
}
 // end main

  // function to sort an array
  void selectionSort( int * const array, const int size )
  {
     int smallest; // index of smallest element

     // loop over size - 1 elements
     for ( int i = 0; i < size - 1; i++ )
     {
        smallest = i; // first index of remaining array
```

```

         // loop to find index of smallest element
         for ( int index = i + 1; index < size; index++ )

            if ( array[ index ] < array[ smallest ] )
               smallest = index;

         swap( &array[ i ], &array[ smallest ] );
      } // end if
   } // end function selectionSort

   // swap values at memory locations to which
   // element1Ptr and element2Ptr point
   void swap( int * const element1Ptr, int * const element2Ptr )
   {
      int hold = *element1Ptr;
      *element1Ptr = *element2Ptr;
      *element2Ptr = hold;
   } // end function swap
```

1.1.3 Efficiency

- The number of comparison made by selection sort is

$$(N-1)+(N-2)+...+1 = N(N-1)/2$$

- Hence, seclection sort would take a time proportional to $O(N^2)$
- Hence, selection sort performs poorly with large numbers.

Exercise: write the template implementation of the provided selection sort code.

1.2 Insertion Sort

- Insertion sort is another iterative sorting algorithm
  - The first iteration of this algorithm takes the second element and, if it is less than the first element, swaps it with the first element (i.e., the program inserts the second element in front of the first element).
  - The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order.
  - At the ith iteration of this algorithm, the first i elements in the original array will be sorted. Note also that the remaining N-i elements are not visted at all.
- The chief virtue of the insertion sort is that it is easy to program; however, it runs slowly.

1.2.1 Example: Sorting 34 56 4 10 77 51 93 30 5 52

1. 34 56 4 10 77 51 93 30 5 52
2. 4 34 56 10 77 51 93 30 5 52
3. 4 10 34 56 77 51 93 30 5 52
4.
5.
6.

1.2.2 Implementation

```cpp
// insertionSort.cpp
 // This program sorts an array's values into ascending order.
 #include <cstdlib>

 #include <iostream>
 using std::cout;
 using std::endl;

 #include <iomanip>
 using std::setw;

int main()
{
   const int arraySize = 10; // size of array a
   int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
   int insert; // temporary variable to hold element to insert

   cout << "Unsorted array:\n";

   // output original array
   for ( int i = 0; i < arraySize; i++ )
      cout << setw( 4 ) << data[ i ];

   // insertion sort
   // loop over the elements of the array
   for ( int next = 1; next < arraySize; next++ )
   {
      insert = data[ next ]; // store the value in the current
         element

      int moveItem = next; // initialize location to place element

      // search for the location in which to put the current element
```

```
         while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
         {
            // shift element one slot to the right
            data[ moveItem ] = data[ moveItem - 1 ];
            moveItem--;
         } // end while

         data[ moveItem ] = insert; // place inserted element into the
            array
      } // end for

      cout << "\nSorted array:\n";

      // output sorted array
      for ( int i = 0; i < arraySize; i++ )
         cout << setw( 4 ) << data[ i ];

      cout << endl;
       system("PAUSE");
     return EXIT_SUCCESS;
   } // end main
```

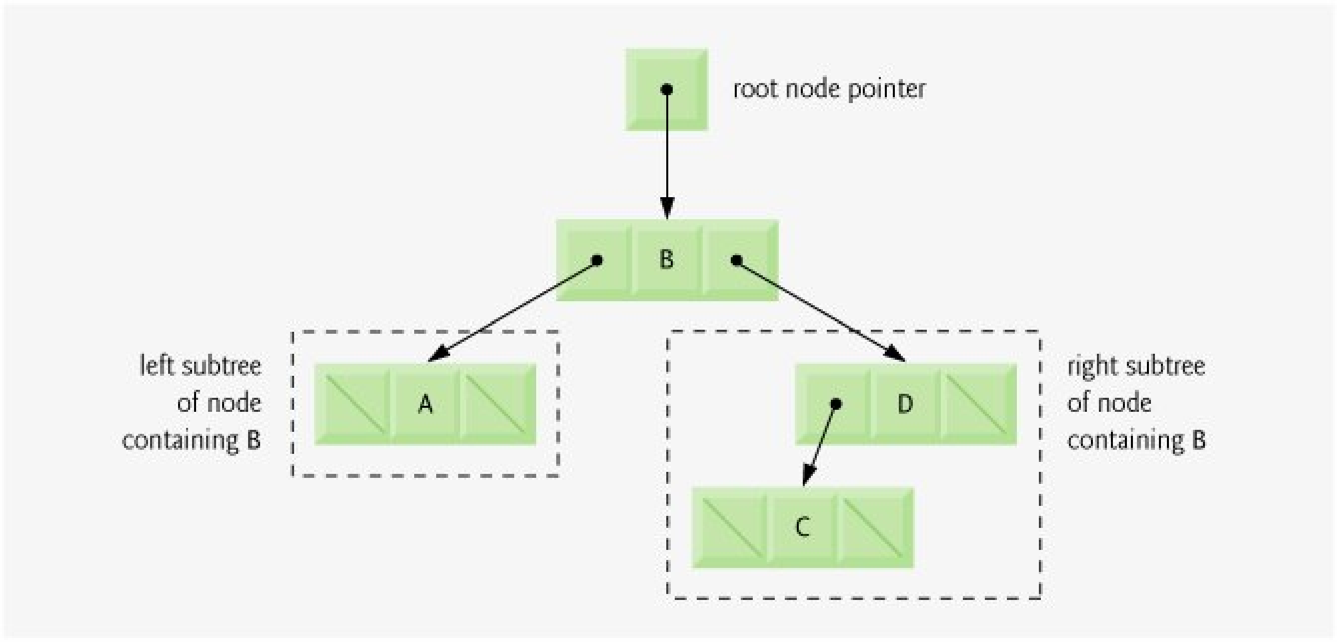
1.2.3 Performance

- Best case: sorted array. N comparisons and no swaps!
- Generally, the algorithm complexity is $O(N^2)$ as selection sort.
  - Insertion sort iterates n times, inserting an element into the appropriate position in the elements sorted so far.
  - For each iteration, determining where to insert the element can require comparing the element to each of the preceding elements in the vector.
  - In the worst case, this will require n comparisons. Each individual repetition statement runs in O(n) time.
  - For determining Big O notation, nested statements mean that you must multiply the number of comparisons. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each O(n) iteration of the outer loop, there will be O(n) iterations of the inner loop, resulting in a Big O of O(n* n) or $O(n^2)$.

1 Trees

1.1 Introduction and basic terminology

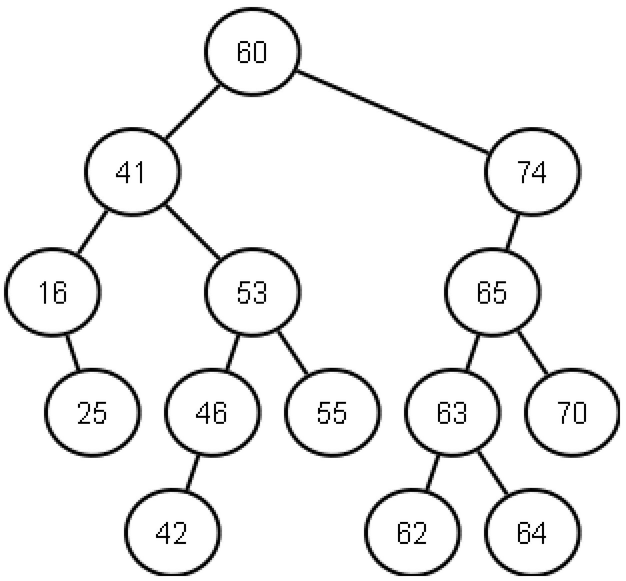
- A tree is a nonlinear, two-dimensional data structure.
- Tree nodes contain two or more links.
- We will focus only on binary trees, trees whose nodes all contain two links (none, one or both of which may be null).



- Tree Terminology
  - The ***root node*** (node B) is the first node in a tree.
  - Each link in the root node refers to a ***child*** (nodes A and D).
  - The ***left child*** (node A) is the root node of the ***left subtree*** (which contains only node A). Similarly, the right child (node D) is the root node of the right subtree (which contains nodes D and C).
  - The children of a single node are called ***siblings*** (e.g., nodes A and D are siblings).
  - A node with no children is called a ***leaf node*** (e.g., nodes A and C are leaf nodes).

1.2 Binary Search Trees

- A binary search tree (with no duplicate node values)
  - the values in any left subtree are less than the value in its parent node
  - the values in any right subtree are greater than the value in its parent node



- Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree
- The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms

1.3 Recurssion

- A ***recursive function*** is a function that calls itself, either directly, or indirectly (through another function)
- Recursive functions are useful in many applications and are of special importance to the implementation of binary search trees.
- A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called ***base case(s)***.

```cpp
  // factorial.cpp
  // Testing the recursive factorial function.

#include <cstdlib>
#include <iostream>
  using std::cout;
  using std::endl;
   #include <iomanip>
  using std::setw;

  unsigned long factorial( unsigned long ); // function prototype

  int main()
  {
     // calculate the factorials of 0 through 10
```

```
      for ( int counter = 0; counter <= 10; counter++ )
         cout << setw( 2 ) << counter << "! = " << factorial( counter )
            << endl;

     system("PAUSE");
     return EXIT_SUCCESS;
  } // end main
  // recursive definition of function factorial
  unsigned long factorial( unsigned long number )
  {
      if ( number <= 1 ) // test for base case
         return 1; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number * factorial( number - 1 );
  } // end function factorial
```

- If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, it typically divides the problem into ***two conceptual pieces***
  - a piece that the function knows how to process and
  - a piece that it does not know how to process
    - This part resembles the original problem, but is slightly simpler or slightly smaller version of the orignal problem.
    - to solve this part, the function performs a ***recursive call***, i.e. the function calls itself again with the simpler problem as a parameter.
  - The function stops the recurssion and return an answer as it reaches a stopping criteria or being called with the input representing the basic problem.
- **Exercise:** Investigate the recursive implementation of list functions including insert, delete, and print. Hint, you may need to use a reference to a pointer

```
void List::linkedListInsert(ListNode *&headPtr, ListItemType
    newItem)
{
  //recursive function implementation.
}
```

  - Which is better in your opinion, the recursive and itterative implementations

1.4 BST Implementation

- Treenode Implementation

```
// Template TreeNode class definition.
#ifndef TREENODE_H
#define TREENODE_H

// forward declaration of class Tree
template< typename NODETYPE > class Tree;

// TreeNode class-template definition
template< typename NODETYPE >
class TreeNode
{
   friend class Tree< NODETYPE >;
public:
   // constructor
   TreeNode( const NODETYPE &d )
      : leftPtr( 0 ), // pointer to left subtree
        data( d ), // tree node data
        rightPtr( 0 ) // pointer to right substree
   {
      // empty body
   } // end TreeNode constructor

   // return copy of node's data
   NODETYPE getData() const
   {
      return data;
   } // end getData function
private:
   TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
   NODETYPE data;
   TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
}; // end class TreeNode

#endif
```

- Tree class header file

```
// Tree.h
// Template Tree class definition.
#ifndef TREE_H
#define TREE_H

```

```
   #include <iostream>
   using std::cout;
   using std::endl;

   #include "Treenode.h"

   // Tree class-template definition
   template< typename NODETYPE > class Tree
   {
   public:
      Tree(); // constructor
      void insertNode( const NODETYPE & );
      void preOrderTraversal() const;
      void inOrderTraversal() const;
      void postOrderTraversal() const;
   private:
      TreeNode< NODETYPE > *rootPtr;

      // utility functions
      void insertNodeHelper( TreeNode< NODETYPE > **, const NODETYPE &
         );
      void preOrderHelper( TreeNode< NODETYPE > * ) const;
      void inOrderHelper( TreeNode< NODETYPE > * ) const;
      void postOrderHelper( TreeNode< NODETYPE > * ) const;
   }; // end class Tree

  #endif
```

1.4.1 Node Insertion

- A node can only be inserted as a leaf node in a binary search tree.
- If the tree is empty, a new TReeNode is created, initialized and inserted in the tree
- If the tree is not empty, the program compares the value to be inserted with the data value starting from the root node until the proper location is found and the node is then inserted.
- The process of creating a binary search tree actually sorts the data. Thus, this process is called the ***binary tree sort***.

1.4.2 Inorder Traversal Algorithm

- Function inOrderTraversal invokes utility function inOrderHelper to perform the inorder traversal of the binary tree. The steps for an inorder traversal are:
  - Traverse the left subtree with an inorder traversal.

- Process the value in the nodei.e., print the node value.
- Traverse the right subtree with an inorder traversal.
- ***the inorder traversal of a binary search tree prints the node values in ascending order.***

1.4.3 Preorder Traversal Algorithm

- Function preOrderTraversal invokes utility function preOrderHelper to perform the preorder traversal of the binary tree. The steps for an preorder traversal are:
  - Process the value in the node.
  - Traverse the left subtree with a preorder traversal.
  - Traverse the right subtree with a preorder traversal.

1.4.4 Postorder Traversal Algorithm

- Function postOrderTraversal invokes utility function postOrderHelper to perform the postorder traversal of the binary tree. The steps for an postorder traversal are:
  - Traverse the left subtree with a postorder traversal.
  - Traverse the right subtree with a postorder traversal.
  - Process the value in the node

- Tree class implementation

```
#inlcude <Tree.h>
 // constructor
  template< typename NODETYPE >
  Tree< NODETYPE >::Tree()
  {
     rootPtr = 0; // indicate tree is initially empty
  } // end Tree constructor

  // insert node in Tree
  template< typename NODETYPE >
  void Tree< NODETYPE >::insertNode( const NODETYPE &value )
  {
     insertNodeHelper( &rootPtr, value );
  } // end function insertNode

  // utility function called by insertNode; receives a pointer
  // to a pointer so that the function can modify pointer's value
  template< typename NODETYPE >
  void Tree< NODETYPE >::insertNodeHelper(
     TreeNode< NODETYPE > **ptr, const NODETYPE &value )
  {
      // subtree is empty; create new TreeNode containing value
      if ( *ptr == 0 )
         *ptr = new TreeNode< NODETYPE >( value );
      else // subtree is not empty
      {
         // data to insert is less than data in current node
         if ( value < ( *ptr )->data )
            insertNodeHelper( &( ( *ptr )->leftPtr ), value );
         else
         {
            // data to insert is greater than data in current node
            if ( value > ( *ptr )->data )
               insertNodeHelper( &( ( *ptr )->rightPtr ), value );
            else // duplicate data value ignored
               cout << value << " dup" << endl;
         } // end else
      } // end else
   } // end function insertNodeHelper

   // begin preorder traversal of Tree
   template< typename NODETYPE >
   void Tree< NODETYPE >::preOrderTraversal() const
   {
      preOrderHelper( rootPtr );
   } // end function preOrderTraversal

   // utility function to perform preorder traversal of Tree
   template< typename NODETYPE >
   void Tree< NODETYPE >::preOrderHelper( TreeNode< NODETYPE > *
      ptr ) const
   {
      if ( ptr != 0 )
      {
         cout << ptr->data << ' '; // process node
         preOrderHelper( ptr->leftPtr ); // traverse left subtree
         preOrderHelper( ptr->rightPtr ); // traverse right
            subtree
      } // end if
   } // end function preOrderHelper

   // begin inorder traversal of Tree
   template< typename NODETYPE >
   void Tree< NODETYPE >::inOrderTraversal() const
   {
      inOrderHelper( rootPtr );
   } // end function inOrderTraversal

   // utility function to perform inorder traversal of Tree
   template< typename NODETYPE >
   void Tree< NODETYPE >::inOrderHelper( TreeNode< NODETYPE > *ptr
      ) const
   {
      if ( ptr != 0 )
      {
         inOrderHelper( ptr->leftPtr ); // traverse left subtree
         cout << ptr->data << ' '; // process node
         inOrderHelper( ptr->rightPtr ); // traverse right subtree
      } // end if
  } // end function inOrderHelper

  // begin postorder traversal of Tree
  template< typename NODETYPE >
  void Tree< NODETYPE >::postOrderTraversal() const
  {
     postOrderHelper( rootPtr );
  } // end function postOrderTraversal

  // utility function to perform postorder traversal of Tree
  template< typename NODETYPE >
  void Tree< NODETYPE >::postOrderHelper(
     TreeNode< NODETYPE > *ptr ) const
  {
     if ( ptr != 0 )
     {
        postOrderHelper( ptr->leftPtr ); // traverse left subtree
        postOrderHelper( ptr->rightPtr ); // traverse right
           subtree
        cout << ptr->data << ' '; // process node
     } // end if
  } // end function postOrderHelper
```