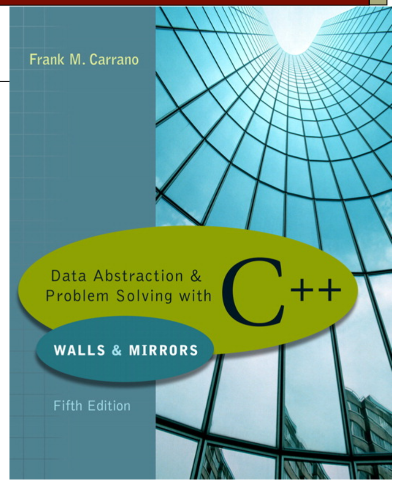
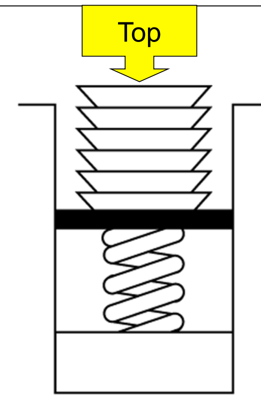
CHAPTER 6
Stacks



Stack of cafeteria dishes



ADT Stack

- create
- destroy
- isEmpty
- push (add a new item to the stack)
- pop (remove from the stack the item that was added most recently)
- getTop (retrieve from the stack the item that was added most recently)

Stack

- Stack of things
- Pile of things
- LIFO: last in first out → Stack
- Only access to the top of the stack

- FIFO: first in first out → Queue

UML diagram for the class **stack**

Stack
top *items*
createStack() *destroyStack()* *isEmpty()* *push()* *pop()* *getTop()*

Stack operations

Stack
top *items*
createStack() *destroyStack()* *isEmpty()* *push()* *pop()* *getTop()*

createStack()
destroyStack()
// Determines whether a stack is empty.
isEmpty()
// Adds an item to the top of a stack.
push(StackItemType newItem, bool & success);
// Removes the top of a stack.
pop(bool & success);
// Retrieves and removes the top of a stack.
pop(StackItemType& stackTop, bool & success);
// Retrieves the top of a stack.
getTop(StackItemType& stackTop, bool & success);

Applications of stack

- checking for balanced braces
- function call and return
- recognizing strings (palindrome) in a language :
  - L = {w$w′, where w′= reverse (w)}
- Algebraic expressions :- postfix, infix, prefix notation
- Evaluating postfix expressions
- converting infix expressions to equivalent postfix expressions
- search problem : depth-first search with backtracking

Checking for balanced braces

- Example strings: { a { b } c }, { a { b c }, { a b } c }
- Requirements for balanced braces
  - Each time } is read, it matches an already encountered {
    - What happens when a } is read, there is no { to be matched?
  - When one reaches the end of a string, each { encountered has been matched
    - What if there is still { to be matched when the end of a string is reached?

Checking for balanced braces

```
read a character in the string
while  (not at the end of the string)
{
    if (the char is '{')
        aStack.push('{')
    else if(the char is '}')
        aStack.pop( )
   read the next character in the string
}
```

Traces of the algorithm that checks for balanced braces

Input string | Stack as algorithm executes

{a{b}c}

Traces of the algorithm that checks for balanced braces

Input string	Stack as algorithm executes 1.	2.	3.	4.	
{a{b}c}	{	{ {	{	(empty)	1. push "{" 2. push "{" 3. pop 4. pop Stack empty ⟹ balanced

{a{bc}

Traces of the algorithm that checks for balanced braces

Input string	Stack as algorithm executes 1.	2.	3.	4.	
{a{b}c}	{	{ {	{	(empty)	1. push "{" 2. push "{" 3. pop 4. pop Stack empty ⟹ balanced
{a{bc}	{	{ {	{		1. push "{" 2. push "{" 3. pop Stack not empty ⟹ not balanced

{ab}c}

Checking for balanced braces

```
aStack.create()
balanced = true
count=0
while(balanced and count < inputString.length()) {
    ch = inputString(count)
    count++
    if(ch == '{')
        aStack.push('{')
    else if (ch == '}')
        if(!aStack.isEmpty())
            aStack.pop()
        else
            balanced = false
}
if (balanced && aStack.isEmpty())
    balanced=true;
else
    balanced = false
```

What is coming

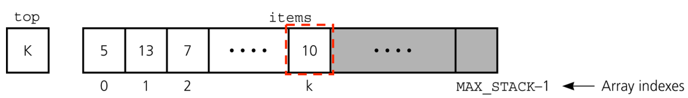
- Implementation of ADT stack
  - Array-based
  - Pointer-based
  - Using ADT list

Implementation of ADT stack

- array-based :- since the data will be stored in statically allocated memory, the compiler generated destructor and copy constructor are sufficient p297
- pointer-based :- destructor, copy constructor and assignment operator overloading is needed
- ADT-List based :- it can be array-based or pointer-based list implementation. Reused of an already implemented class saves time
- Standard template library class **stack**

Implementations of the ADT stack that use
(a) an array
(b) a linked lis
(c) an ADT lis

An array-based implementation



Header file StackA.h for the ADT stack. Array-based implementation.

```
class Stack {
public:
    Stack();
    bool isEmpty() const;
    void push(StackItemType newItem, bool & success);
    void pop(bool & success);
    void pop(StackItemType& stackTop, bool & success);
    void getTop(StackItemType& stackTop, bool & success);
private:
    StackItemType items[MAX_STACK]; // array of stack items
    int top; // index to top of stack
}; // end class
```

Array-based Implementation: StackA.cpp

```
#include "StackA.h"   // Stack class specification file
Stack::Stack(): top(-1)
{ } // end default constructor

bool Stack::isEmpty() const {
    return top < 0;
} // end isEmpty

void Stack::push(StackItemType newItem. bool & success) {
    success = true;
    // if stack has no more room for another item
    if (top >= MAX_STACK-1)
        success = false;
    else {
        ++top;
        items[top] = newItem;
    } // end if
} // end push
```

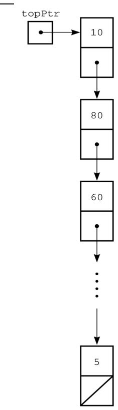
Array-based Implementation: StackA.cpp

```
void Stack::pop(bool & success) {
    success = true;
    if (isEmpty())
        success = false;
    else
        --top; // stack is not empty, pop top
} // end pop

void Stack::pop(StackItemType& stackTop, bool &success) {
    success = true;
    if (isEmpty())
        success = false;
    else  { // stack is not empty, retrieve top
        stackTop = items[top];
        --top; // pop top
    } // end if
} // end pop

void Stack::getTop(StackItemType& stackTop, bool & success) const  {
    success = true;
    if (isEmpty())
        success = false;
    else  // stack is not empty; retrieve top
        stackTop = items[top];
} // end getTop
```

What is coming

- Pointer-based stack
- Search problem

A pointer-based implementation



Pointer-based Implementation: StackP.h

```
class Stack
{
public:
      Stack();
      Stack(const Stack& aStack);
      ~Stack();
      // stack operations:
      bool isEmpty() const;
      void push(StackItemType newItem);
      void pop(bool & success);
      void pop(StackItemType& stackTop, bool & success);
      void getTop(StackItemType& stackTop, bool & success) const;
private:
      struct StackNode {  // a node on the stack
            StackItemType item;
            StackNode *next;
      }; // end struct
      StackNode *topPtr; // pointer to first node in the stack
}; // end Stack class
```

Pointer-based Implementation: StackP.cpp

```
#include "StackP.h"
#include <cstddef> // for NULL
#include <cassert>  // for assert
Stack::Stack( ) : topPtr(NULL) {  } // end default constructor

Stack::Stack(const Stack& aStack) {
      if (aStack.topPtr == NULL)
            topPtr = NULL; // original list is empty
      else {
            // copy first node
            topPtr = new StackNode;
            assert(topPtr != NULL);
            topPtr->item = aStack.topPtr->item;
            // copy rest of list
            StackNode *newPtr = topPtr;
            for (StackNode *origPtr = aStack.topPtr->next;
                        origPtr != NULL; origPtr = origPtr->next)    {
               newPtr->next = new StackNode;
               assert(newPtr->next != NULL);
               newPtr = newPtr->next;
               newPtr->item = origPtr->item;
            } // end for
            newPtr->next = NULL;
      } // end if
} // end copy constructor
```

Pointer-based Implementation: StackP.cpp

```
Stack::~Stack() {
    while (!isEmpty())
        pop();
} // end destructor

bool Stack::isEmpty() const {
    return topPtr == NULL;
} // end isEmpty
```

Pointer-based Implementation: StackP.cpp

```
void Stack::push(StackItemType newItem, bool & success) {
    success = true;
    // create a new node
    StackNode *newPtr = new StackNode;
    if (newPtr == NULL)
        success = false;
    else { // allocation successful; set data portion of new node
        newPtr->item = newItem;
        newPtr->next = topPtr;
        topPtr = newPtr;
    } // end if
} // end push
```

Pointer-based Implementation: StackP.cpp

```
void Stack::pop(bool & success) {
    success = true;
    if (isEmpty())
        success = false;
    else { // stack is not empty; delete top
        StackNode *temp = topPtr;
        topPtr = topPtr->next;
        temp->next = NULL; // safeguard
        delete temp;
    } // end if
} // end pop

void Stack::pop(StackItemType& stackTop, bool & success) {
    success = true;
    if (isEmpty())
        success = false;
    else { // stack is not empty; retrieve and delete top
        stackTop = topPtr->item;
        StackNode *temp = topPtr;
        topPtr = topPtr->next;
        temp->next = NULL; // safeguard
        delete temp;
    } // end if
} // end pop
```

Pointer-based Implementation: StackP.cpp

```
void Stack::getTop(StackItemType& stackTop, bool & success) const {
    success = true;
    if (isEmpty())
        success = false;
    else
        // stack is not empty; retrieve top
        stackTop = topPtr->item;
    } // end getTop
```

What is coming

- Infix notation

Arithmetic expression

2 * (3 + 4)

- Infix notation

2 3 4 + *

- Postfix notation

* 2 (+ 3 4)

- Prefix notation

How do we evaluate this?
1) Operator precedence
2) Left to right

How do we evaluate this?
1) Left to right
2) Use stack

How do we evaluate this?
1) Like function call

The action of a postfix calculator
when evaluating the expression 2 * (3 + 4)
Which is equivalent to 2 3 4 + *

parenthesis

Store operands in the stack

Key entered	Calculator action	k (bottom to top)

Evaluating postfix expressi

2 * (3 + 4)
→
2 3 4 + *

```
for each ch in the string{
    if (ch is an operand)
        push operand onto the stack
    else if (ch is an operator) {
        // evaluate and push the result
        op2 = stack.pop()
        op1 = stack.pop()
        result = op1 op op2
        stack.push(result)
    }
```

How can we convert infix notation to postfix notation?

2 * (3 + 4) ➔ 2 3 4 + *

- Infix notation to Postfix notation
- Use stack to store operators

A trace of the algorithm that converts the infix expression a - (b + c * d)/e to postfix form a b c d * + e / -

ch	Stack (bottom to top)	postfixExp

converting infix expressions to equivalent postfix expressions

infix: a - (b + c * d)/e
postfix: a b c d * + e / -

```
string infix, postfix;
read infix
for (each ch in the infix expression) {
      switch (ch) {
        case operand:        postfix += ch;       break;
        case '(':            aStack.push(ch);     break;
        case ')':     while(aStack.top ( ) != '(')
                             postfix += aStack.pop ( );
                    aStack.pop();  // pop '('
                    break;
        case operator:       while(!aStack.isEmpty( ) and aStack.top( ) != '(' and
                               precedence(ch) <= precedence(aStack.top()))
                                    postfix += aStack.pop();
                             aStack.push(ch);
                             break;
      }
}
while (!aStack.isEmpty())
      postfix += aStack.pop();
```

)	– (+	abcd*
	– (	abcd*+
	–	abcd*+

What is coming

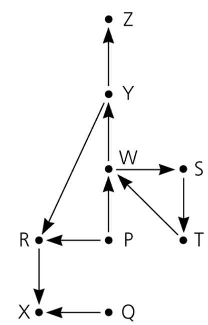
- Search problem solved using stack
- OLA 7

A Search Problem

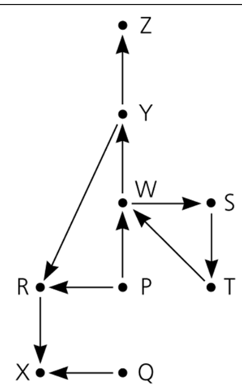
- The EastWest airline company wants you to help them develop a program that generates flight itineraries for customer requests to fly from some origin city to some destination city.

Input data

- Cities
  - nodes in the graph
- Flight records
  - *178 Albuquerque Chicago 250*
  - Flight #, origin, destination, cost
  - Links in the graph
- Representation: directed graph



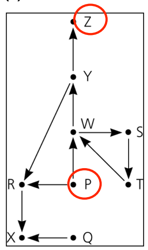
Flight map for HPAir



To find flight itinerary origin to destination

```
aStack.create()
aStack.push(origin)
while(not found) {
    if(need to backtrack from the city on top of stack)
        aStack.pop()
    else {
        select a destination for a flight from city on top
        aStack.push(destination)
    }
}
```

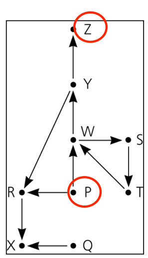
The stack of cities as you travel
(a) from *P*;
(b) to *R*;
(c) to *X*;
(d) back to *R*;
(e) back to *P*;
(f) to *W*

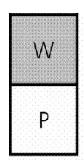


Three possible outcomes

- Eventually reach the destination city ☺
- Reach a city from which there are no departing flight ☹
- Go around in circles ☹ ☹

The stack of cities
(a)allowing revisits





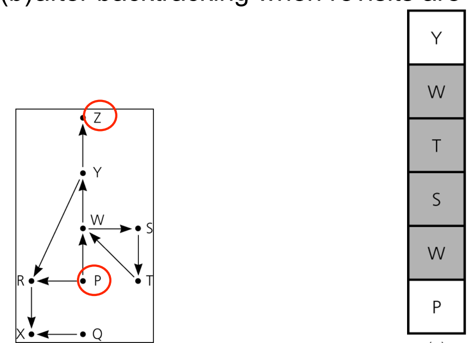
Solving the problem using a stack

- What is in the stack?
  - Sequence of flights currently under consideration
  - Top of the stack is the currently visiting city
  - Bottom of the stack is the origin city
- How to find the path from the origin city to the destination city?
  - from the bottom to the top
- What do you do when you reach a dead-end?
  - No flight out of that city
- What happens if there is a cycle?

You never want to visit a city that the search has already visited

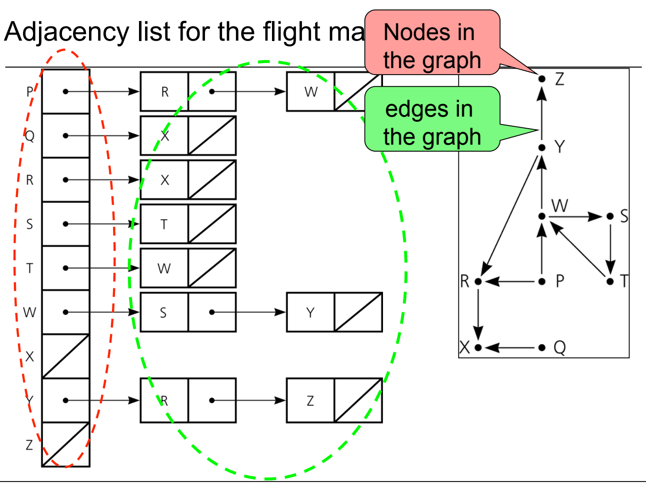
- Backtrack whenever there are no more unvisited cities to fly to
  - Visited city is still in the stack
  - Visited city is not in the stack because you backtracked from it
- Mark the visited city and choose the next city which is unmarked (not visited) and adjacent to the city on top of the stack

The stack of cities (a)allowing revisits and (b)after backtracking when revisits are not allowed



(a)

Adjacency list for the flight map



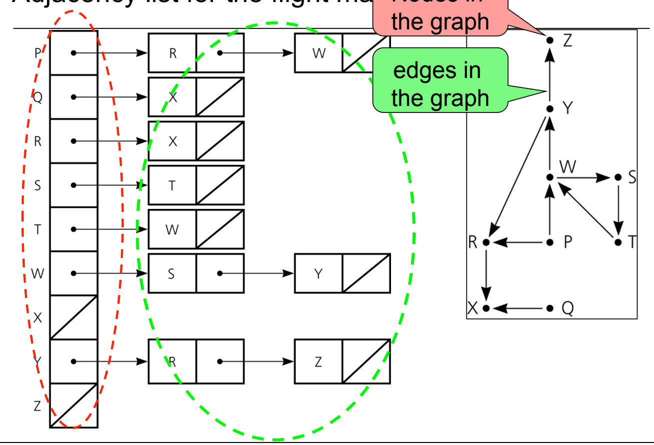
Revised search algorithm (p323)

```
aStack.create()  and clear marks on all cities
aStack.push(origin) and mark origin as visited
while(!aStack.isEmpty( ) and destin != top of aStack) {
    if (no flight exist from city on top of aStack to unvisited cities)
        aStack.pop()
    else {
        select an unvisited city (C) for a flight from city on top of
        the stack
        aStack.push(C)
        mark C as visited
    }
}
if (aStack.isEmpty( ))          return false  // no path exist
else     return true  // path found
```

IsPath search algorithm in FlightMap class (p326)

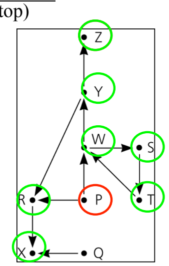
```
bool Map::isPath (int originCity, int destinationCity) {
  Stack aStack;
  int   topCity, nextCity;   bool  success;
  unvisitAll ( ); // clear marks on all cities
  aStack.push (originCity);
  markVisited (originCity);
  aStack.getTop (topCity);
  while (!aStack.isEmpty( ) && (topCity != destinationCity))   {
      success = getNextCity(topCity, nextCity);
      if (!success)
           aStack.pop(); // no city found; backtrack
      else  {  // visit city
           aStack.push(nextCity);
           markVisited(nextCity);
      } // end if
      aStack.getTop(topCity);
  } // end while
  return  !(aStack.isEmpty())  // if stack is empty, no path exist
} // end isPath
```

Adjacency list for the flight ma



A trace of the search algorithm, given the flight map

Action	Reason	Contents of Stack (Bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z



What is coming

- Search problem: recursive solution

A recursive solution

Is path from origin to destin?	isPath(origin, destin)
Select a city C adjacent to origin	getNextCity(origin)
Fly from origin to C	Push C to stack
If C is the destination	If C == destin
Stop	Return (base case)
Else	Else
Fly from C to destin..	isPath(C, destin)

Three possible outcomes

- Eventually reach the destination city ☺
- Reach a city from which there are no departing flight ☹
- Go around in circles ☹ ☹

A recursive solution

Is path from origin to destin?

Mark origin as visited
Select a city C adjacent to origin
Fly from origin to C
If C is the destination
    Stop
Else
    Fly from C to destin..

isPath(origin, destin)

Mark origin as visited
getNextCity(origin)
Push C to stack
If C == destin
    Return (base case)
Else
  For each unvisited adjacent city
    isPath(C, destin)

Recursive IsPath search algorithm (p329)

```
bool Map::isPath(int originCity, int destinationCity) {
  int nextCity;   bool success, done;
  markVisited(originCity);
  // base case: the destination is reached
  if (originCity == destinationCity)
    return true;
  else // try a flight to each unvisited city
  {  done = false;
    success = getNextCity(originCity, nextCity);
    while (success && !done)
    {  done = isPath(nextCity, destinationCity);
       if (!done)
         success = getNextCity(originCity, nextCity);
    } // end while
    return done;
  } // end if
} // end isPath
```

```
bool Map::isPath (int originCity, int destinationCity, stack<FlightNode>& aStack) {
  int  nextCity;
  bool success, done;
  markVisited(originCity);
  // base case: the destination is reached
  if (originCity == destinationCity)
            return true;
  else  {   // try a flight to each unvisited city
        done = false;
        FlightNode flightInfo;
        success = getNextCity(originCity, nextCity, flightInfo);
        while (success && !done)  {
              done = isPath(nextCity, destinationCity, aStack);
              if (!done)
                success = getNextCity(originCity, nextCity, flightInfo);
              else {
                 cout << " from " << cities[nextCity] << endl;
                 aStack.push(flightInfo);
              }
        } // end while
    return done;
  } // end if
} // end isPath
```