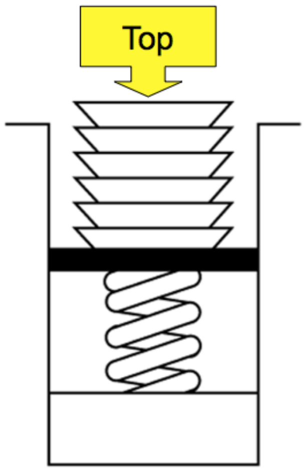
CSCI 3110 STL Stack



Characteristic: **Last In First Out (LIFO)**
Operations:

- Create an empty stack
- Destroy a stack
- Determine whether a stack is empty -- **empty()**
- Add a new item to the stack – **push(ItemType newItem)**
- Remove from the stack the item that was added most recently -- **pop()**
- Retrieve the item that was added most recently -- **top()**

How to create an empty stack using the C++ Stack Container

```
stack <string>   stringStack;
stack<int>   intStack;
```

Applications of Stack

(1) Read characters and correct with backspace: reads the input line, for each character read, either enter it into stack S, if it is '←', correct the content of S

```
void ReadAndCorrect(stack <char>& aStack)   {
        char  ch;
        cin.get(ch);  // read in a character

        while (ch != '\n')   {  // newChar is not the end of line symbol
                if (ch != '\b')    // newChar is not backspace character '←'
                        aStack.push(ch);
                else if  (!aStack.empty())
                        aStack.pop();

                cin.get(ch);
        }
}
```

(2) Display the content of a stack : directly popping out the content of stack will display the letters in the word in reverse order

```
void DisplayBackward(stack <char>& aStack)  {
        char ch;
        while (!aStack.empty())   {
                ch = aStack.pop();
                cout << ch;
        }
}
```

?? How to write out the content of the stack in the original order when they were read?

```
void DisplayForward ( stack <char>& aStack)           {
        aStack <char>   auxStack;          char ch;
        while (!aStack.empty())    {
                ch=aStack.top();
                aStack.pop();
                auxStack.push(ch);
        }

        DisplayBackward(auxStack);
}
```

?? What is the content of the stack after executing this function?

?? How to count the number of items stored in a stack and keep the content of the stack unchanged after the operation??

(3) Checking for balanced braces

- each time a '{' is encountered, push it onto the stack
- each time a '}' is entered, it is matched to an already encountered '{', pop stack
- **Balanced :** when reaching the end of the string, all the '{' has been matched against (stack is empty)
- **NOT balanced** :
  1. when a '}' is entered, there is no existing '{' to match, OR
  2. when reaching the end of the string, there are still some '{' not being matched (stack not empty)

```
void CheckBalanced(string program)  {
        int  index = 0;
        bool balanced = true, success = false;
        stack  <char> braces;
        while (balanced && index < strlen(program))   {
                ch = program [index];
                index ++;

                if (ch == '{')
                        braces.push(ch);
                else if (ch == '}')    {
                        if ( ! braces.empty())
                                braces.pop();
                        else
                                balanced = false;
                }
        }

        if (balanced && braces.empty())
                cout << "The braces in this program are balanced." << endl;
        else
                cout << "Syntax error: Braces are NOT balanced." << endl;
}
```

Extend this program to work with checking for balanced [] and balanced ().

(4) Arithmetic Expression Evaluation

Infix notation **2 * (3 + 4)** ➔ Postfix notation **2 3 4 + ***

How to evaluate postfix expressions? Pseudocode:

```
stack <char> aStack;
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 = aStack.top()
        aStack.pop();
        op1 = aStack.top()
        aStack.pop()
        result = op1 op op2
        aStack.push(result)
    }
```

Practice: What are the values of these postfix expressions:

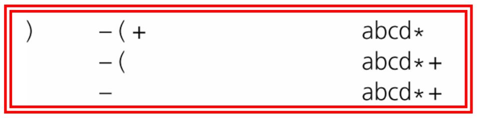
- 50 10 – 40 + 30 20 - *
- 30 20 20 10 - - * 10 +

(5) Convert infix expression to postfix expression. Pseudocode:

```
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.top ( );
                            aStack.pop();
                        }
                        aStack.pop();  // pop '('
                        break;
     case operator:     while(!aStack.isEmpty( ) and aStack.top( ) != '(' and
                          precedence(ch) <= precedence(aStack.top())) {
                            postfix += aStack.top();        aStack.pop();
                        }
                        aStack.push(ch);
                        break;
    }
}
while (!aStack.isEmpty())
    postfix += aStack.pop();
```

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

postfix: a b c d * + e / -



)	– (+	abcd∗
	– (	abcd∗ +
	–	abcd∗ +

Practice: What are the corresponding postfix expressions for the following infix expressions?

1. Infix: a * (b * c – d) + e
2. Infix: 5 * ((4 – x)* (6 + y – x))/(x – y)

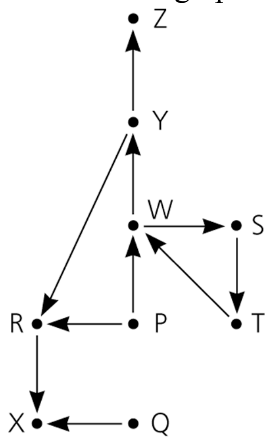
(6) Search Problem

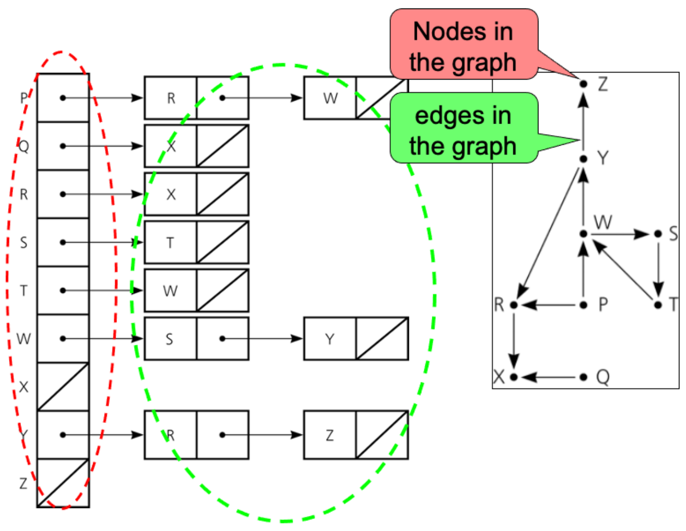
Flight scheduler (list, stack)

- Create a flight map based on the flights served by the airline company
  - The nodes in the graph represent the cities
  - Flight records are in the forms of

178	*Albuquerque*	*Chicago*	*250*
Flight#	origin	destination	cost

- Flight map is a directed graph





- Question: what is the flight itinerary to go from city X to Z?
- Answer: Use a stack for the search
  - What is in the stack?
    - Sequence of flights currently under consideration

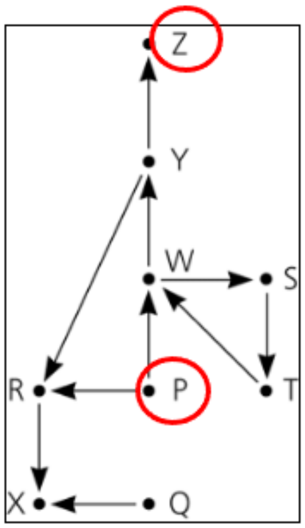
        - Top of the stack is the currently visiting city
        - Bottom of the stack is the origin city
    - How to find the path from the origin city to the destination city?
        - from the bottom to the top

<u>Version 1 pseudocode:</u>

```
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 city for a flight from city on top of the stack
        aStack.push(destination)
    }
}
```

Assuming we are trying to go from city P to city Z, here is a partial trace of the cities one "visits" during the process of deriving the flight itinerary

(a) from *P*;
(b) to *R*;
(c) to *X*;
(d) back to *R*;
(e) back to *P*;
(f) to *W*
(g) ...



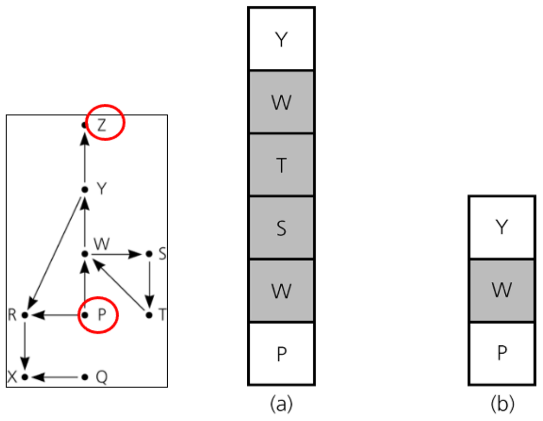
Assuming we are trying to go from city P to city Z

- 3 possible outcomes of this code
    - Eventually reach the destination city ☺
    - Reach a city from which there are no departing flight ☹
    - Go around in circles ☹ ☹
- Need to solve the problems:
    1. What do you do when you reach a dead-end? (no flight out of that city)
    2. What happens if there is a cycle?
- <u>Solution 1</u>:
    - 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

- <u>Solution 2:</u> keep track of which cities have already been visited, and not allow revisit a city during the search process.

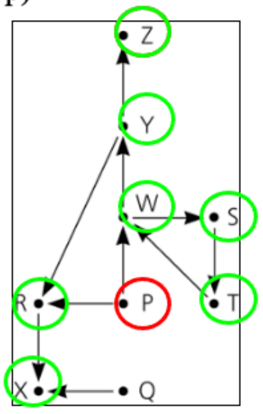
The stack of cities
(a)allowing revisits and
(b)after backtracking when revisits are not allowed



Revised code:

```
aStack.create()  and clear marks on all cities
aStack.push(origin) and mark origin as visited
while(!aStack.isEmpty( ) and destination != top of aStack) {
if (no flight exists 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;
      }
}  // end while
if (aStack.isEmpty( ))          return false;  // no path exist
else      return true;  // path found
```

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



```
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
```

Discussion:

When to use Adjacency Matrix and when to use Adjacency List to represent graph?

If the connection between nodes is dense, use adjacency matrix.
If the connection among nodes is sparse, i.e., an adjacency matrix becomes a sparse matrix, it is better to use adjacency list.

What is a sparse matrix?
2/3 of the matrix elements does not have value, or have value 0. Or, on average each node has connection to less than 1/3 other nodes in the graph.