

Stacks

Stack

- Last in First Out

```
#include <stack>
// Stack operations
stack<T> stackExample;
stackExample.push();
stackExample.pop();
stackExample.top();
stackExample.empty();
stackExample.size();
// with constructor & destructor
```

Stack Model - LIFO

- The top allows access to the top of the "Stack" - Any list implementation could be used to make a stack - Operating on one end - Vector/List ADTs - push_front()/pop_front() - push_back()/pop_back()

Stack Uses

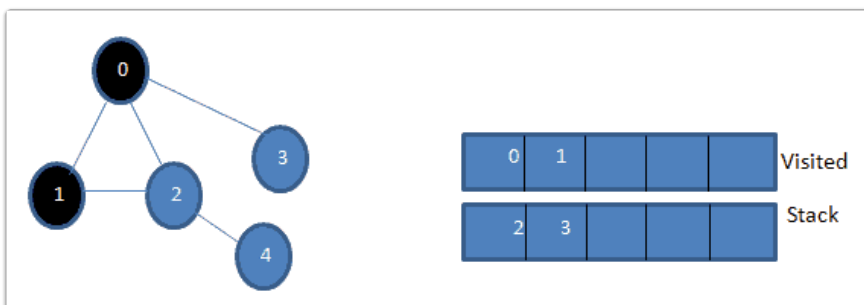
- Depth-first search/backtracking - Evaluating postfix expressions - Converting infix to postfix - Function calls (runtime stack) - Recursion

Runtime Stack

- Static - Executable code - Global variables - Stack - Push for each function call - Pop for each function return - Local variables - Heap - Dynamically allocated memories - new and delete

Depth First Expanded

- If there is an unlisted neighbor go there - Retreat along the path to find unlisted neighbor, it cannot go deeper - If there is a path from start to goal, DFS finds one such path - Discover a path from **start** to the goal - Start from Node start stop if Node reaches goal



```
// Depth First Search
DFS() {
    stack<location> S;
    //Mark the start location as visited
        S.push(start);
        while (!S.empty()) {
            t = S.top();
            if (t == goal) Success(S);
            if (// t has unvisited neighbors) {
                //Choose an unvisited neighbor n
                // mark n visited;
                S.push(n);
            } else {
                BackTrack(S);
            }
        }
        Failure(S);
}

/*
                                Another Implementation Of DFS
*/

BackTrack(S) {
    while (!S.empty() && S.top() has no unvisited neighbors) {
        S.pop();
    }
}

Success(S) {
    // print success
    while (!S.empty()) {
        output(S.top());
        S.pop();
    }
}

Failure(S) {
    // print failure
    while (!S.empty()) {
        S.pop();
    }
}
```

```
    }  
}
```

Postfix Expressions

- Use a stack of tokens - Repeat - If operand, push onto the stack - If operator - pop operands off the stack - evaluate operator on operands - push the result onto the stack - Until expression is read - Return top of the stack

```
Evaluate(postfix expression) {  
  
    // use stack of tokens;  
    while(// expression is not empty) {  
        t = next token;  
        if (t is operand) {  
            // push onto the stack  
        } else {  
            // pop operands for t off stack  
            // evaluate t on these operands  
            // push the result onto the stack  
        }  
    }  
    // return top of stack  
}
```

Postfix Visualized

$$3 \ 10 \ 5 \ + \ * = 3*(10+5) = 45$$

