TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Data structures and Algorithms Basic Lab

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Course outline

2

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chapter 4. Stack and Queue

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents

1. **Stack**
2. Queue

1. Stack

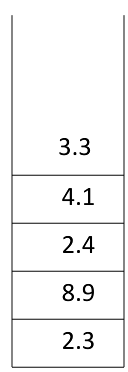
1. Implementation of stack using the linked list
2. An application of stack

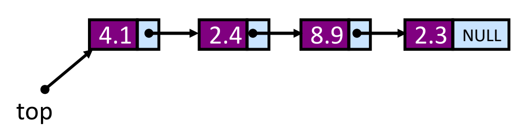
1. Stack

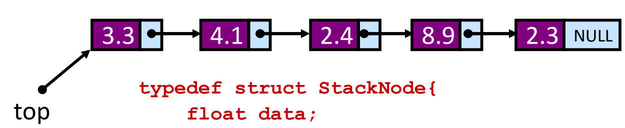
1. **Implementation of stack using the linked list**
2. An application of stack

Implementing a Stack: using a linked list

- Store the items in the stack in a linked list
- The top of the stack is the head node, the bottom of the stack is the end of the list
- *push* by adding to the front of the list
- *pop* by removing from the front of the list



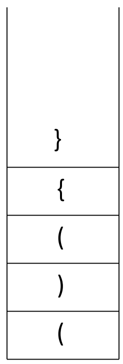


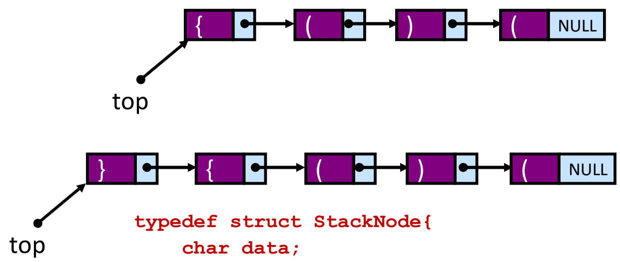



```
typedef struct StackNode{
    float data;
    struct StackNode *next;
} StackNode;
typedef struct {
    StackNode *top;
}Stack;
```

Exercise

- Implement a stack by using a linked list: each element in the stack is a character selected among (,), {, }, [,]






```
typedef struct StackNode{
    char data;
    struct StackNode *next;
} StackNode;
typedef struct {
    StackNode *top;
}Stack;
```

Operations

1. Init Stack:

   `void init(Stack* s);`

2. Check empty:

   `int isEmpty(Stack s);`

3. Insert a new item into stack (Push): *insert a new element with data=new_data at the top of stack* `int push(Stack* s, char new_data);`
return 1 if operation push is successful, otherwise return 0

4. Remove an item from stack (Pop): *remove and return the item at the top of stack:*

   `char pop(Stack* s);`

5. Destroy stack

   `void destroy(Stack* s);`

Init stack

```
void init(Stack *s)
{
    s->top = NULL;
}
```

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Check empty

```
int isEmpty(Stack s)
```

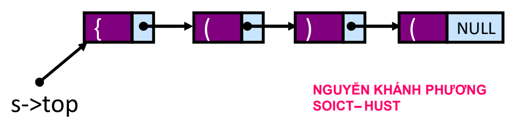
reurn 1 if stack is empty; otherwise return 0

```
int isEmpty(Stack s)
{
    return s.top==NULL;
}
```

Push

Need to do the following steps:
(1) Create new node: allocate memory and assign data for new node
(2) Link this new node to the top (head) node
(3) Assign this new node as top (head) node

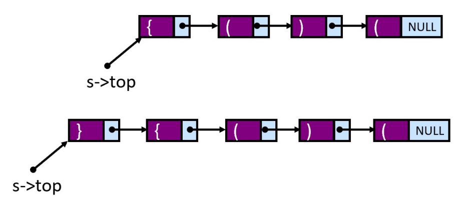
```
int push(Stack *s, char new_data) {
  StackNode *node;
  node = (StackNode *)malloc(sizeof(StackNode)); //(1)
  if (node == NULL) {// overflow: out of memory
    printf("Out of memory");return 0;
  }
  node->data = new_data;    //(1)
  node->next = s->top;      //(2)
  s->top = node;            //(3)
  return 1;
}
```



NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Pop

1. Check whether the stack is empty
2. Memorize address of the current top (head) node
3. Memorize data of the current top (head) node
4. Update the top (head) node: the top (head) node now points to its next node
5. Free the old top (head) node
6. Return data of the old top (head) node




```
char pop(Stack *s) {
    char data;
    StackNode *node;
    if (isEmpty(*s))        //(1)
      return NULL;       // Empty Stack, can't pop
    node = s->top;           //(2)
    data = node->data;       //(3)
    s->top = node->next;     //(4)
    free(node);              //(5)
    return data;             //(6)
}
```

Destroy stack

```
void destroy(Stack *s)
{
    while (!isEmpty(s)) pop(s);
    free(s);
}
```

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

2. Stack

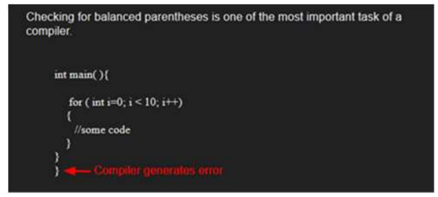
1. Implementation of stack using the linked list
2. **An application of stack:** Parentheses Matching

Application of stack: Parentheses Matching

Check for balanced parentheses in an expression:

Given an expression string expression, write a program to examine whether the pairs and the orders of "{","}","(",")","[","]" are correct in expression.

For example, the program should print true for expression = "[()]{}{[()()]()}" and false for expression = "[(])"



Algorithm:

1) Declare a character stack S.
2) Now traverse the expression string expression
   a) If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
   b) If the current character is a closing bracket (')' or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3) After complete traversal, if there is some starting bracket left in stack then "not balanced"

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Application of stack: Parentheses Matching

Algorithm ParenMatch(*X*,*n*):

Input: Array *X* consists of *n* characters, each character could be either parentheses, variable, arithmetic operation, number.

Output: **true if** parentheses in an expression are **balanced**

```
S = stack empty;
for i=0 to n-1 do
    if (X[i] is a starting bracket)
        push(S, X[i]);    // starting bracket ('(' or '{' or '[') then push it to stack
    else
        if (X[i] is a closing bracket)  // compare X[i] with the one currently on the top of stack
            if isEmpty(S)
                    return false {can not find pair of brackets}
            if (pop(S) not pair with bracket stored in X[i])
                    return false {error: type of brackets}
if isEmpty(S)
    return true {parentheses are balanced}
else  return false {there exist a bracket not paired}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Application of stack: Parentheses Matching

```
bool isPair(char a, char b)
{
    if(a == '(' && b == ')') return true;
    if(a == '{' && b == '}') return true;
    if(a == '[' && b == ']') return true;
    return false;
}
```

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

```
int solve(char  *x, int n)
{
  Stack S; init(&S);
  for(int i = 0; i <= n-1; i++)
  {
     if(x[i] == '[' || x[i] == '(' || x[i] == '{')  push(&S, x[i]);
     else
     if(x[i] == ']' || x[i] == ')' || x[i] == '}') {
        if (isEmpty(S)) return false;
        else {
          char c = pop(&S);
          if(!isPair(c,x[i]))  return false;
        }
    }//end if
 }//end for
 return isEmpty(S);
}
int main() {
  bool ok = solve("[({})]()",8);
  if (ok) printf("Parentheses in the expression are balanced");
  else printf("Not balanced");
}
```

Contents

1. Stack
2. **Queue**

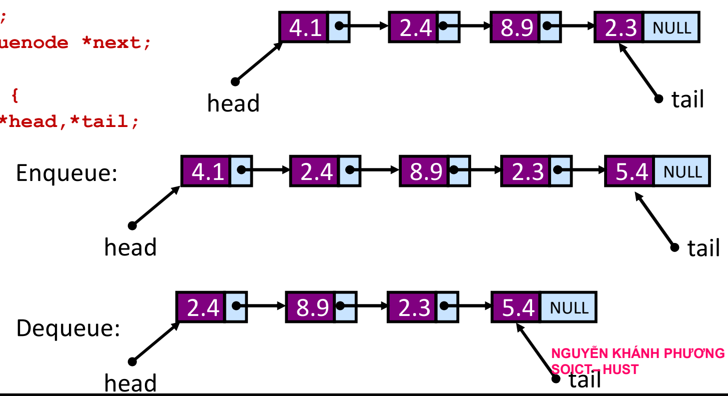
2. Queue

1. **Implementation of queue using the linked list**
2. An application of queue

Implementing a Queue: using a linked list

- Store the items in the queue in a linked list
- The top of the queue is the head node, the bottom of the queue is the end of the list
- *Enqueue* by adding a new element to the front of the list
- *Dequeue* by removing the last element from the list

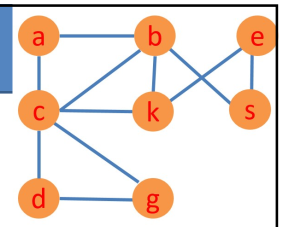
```
typedef struct Queuenode{
    float item;
    struct Queuenode *next;
}Queuenode;
typedef struct {
    Queuenode *head,*tail;
}Queue;
```



2. Queue

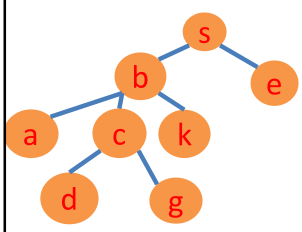
1. Implementation of queue using the linked list
2. **An application of queue**

Breadth First Search



G=(V, E)

- Given
  - a graph G=(V,E) – set of vertices and edges
  - a distinguished source vertex s
- Breadth first search systematically explores the edges of G to discover every vertex that is reachable from s.
- For any vertex *v* reachable from *s*, the path in the breadth first tree corresponds to the shortest path in graph G from *s* to *v*.



BFS(s) tree

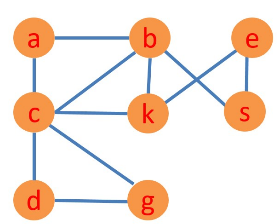
BFS creates a BFS tree containing s as the root and all vertices that is reachable from s

Adjacency list of s

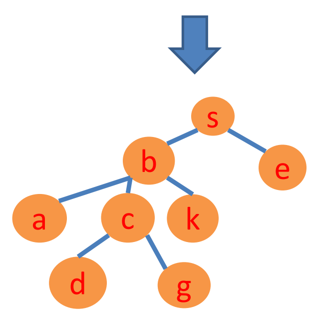
- From s: can go to b and e. Visit them and insert them into queue: Q = {b, e}
- Dequeue (Q): remove b out of Q, then Q = {e}
  - From b: can go to a, c, k, s. But s was visited, so we visit only a, c, k; and insert them into queue: Q = {e, a, c, k}
- Dequeue(Q): remove e out of Q, then Q = {a, c, k}
  - From e: can go to k, s. But all of them were visited.
- Dequeue(Q): remove a out of Q, then Q = {c, k}
  - From a: can go to b, c. But these vertices were all visited.
- Dequeue(Q): remove c out of Q, then Q = {k}.
  - From c: can go to a, b, d, g, k. But a, b, k were visited, so we visit only d, g; and insert them into queue: Q = {k, d, g}
- Dequeue(Q): remove k out of Q, then Q = {d, g}.
  - From k: can go to b, c, e. But these vertices were all visited.
- Dequeue (Q): remove d from Q, then Q = {g}
  - From d: can go to c, g. But these vertices were all visited.
- Dequeue(Q): remove g from Q, then Q = empty
  - From g: can go to d, c. But these vertices were all visited.
- Q is now empty. All vertices of the graph were visited. Algorithm is finished

Breadth-first Search

```
BFS(s)
// Breadth first search starts from vertex s
    visited[s] ← 1; //visited
    Q ← ∅; enqueue(Q,s); // insert s into Q
    while (Q ≠ ∅)
    {
        u ← dequeue(Q);  // Remove u from Q
        for  v ∈Adj[u]
            if (visited[v] == 0) //not visited yet
            {
                visited[v] ← 1; //visited
                enqueue(Q,v)   // insert v into Q
            }
    }

(*Main Program*)
main ()
    for  s  ∈ V  // Initialize
        visited[s] ← 0;

    for  s  ∈ V
      if (visited[s]==0)  BFS(s);
```

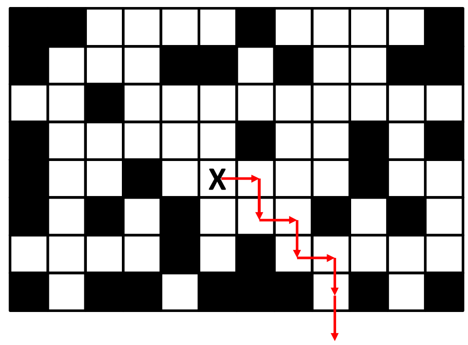


G=(V, E)



MAZE problem

- A Maze is represented by a 0-1 matrix $maze_{N\text{x}M}$ in which maze[i][j] = 1 means cell (i,j) is an obstacle, maze[i][j] = 0 means cell (i,j) is free.
- From a free cell, we can go up, down, left, or right to an adjacent free cell.
- Compute the minimal number of steps to escape from a Maze from a given start cell (i_0, j_0) within the Maze:
  - Answer: Can escape or not ?
  - Answer: If can escape, print the way to escape.

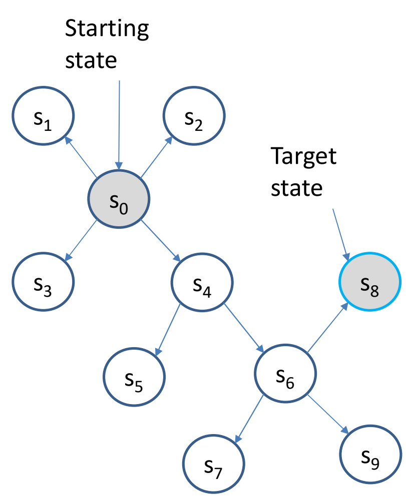


Escape the Maze after 7 steps

27

MAZE problem

- A state of the problem is represented by (r,c) which are respectively the row and column of a position
- Search Algorithm:
  - Push the starting state into the queue
  - Loop
    - Pop a state out of the queue, generate neighbor states and push them into the queue if they were not generated so far
  - The algorithm terminates when the target stated is generated



MAZE problem

```
typedef struct Node{
    int row,col;// the index of row and column in the maze of the node
    struct Node* next; // pointed to the next node in the queue
}Node;
```

```
Node* makeNode(int row, int col)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->row = row; node->col = col; node->next = NULL;
    return node;
}
```

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

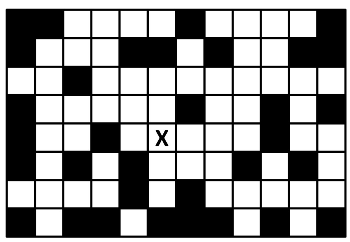
MAZE problem

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

Node* head, *tail;
int maze[MAX][MAX];
int n,m, r0, c0;
int visited[MAX][MAX];

const int dr[4] = {1,-1,0,0};
const int dc[4] = {0,0,1,-1};
```

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

MAZE problem

```
void input() //read maze
{
    FILE* f = fopen("maze.txt","r");
    fscanf(f,"%d%d%d%d",&n,&m,&r0,&c0);
    for(int i = 1; i <= n; i++)
        for(int j =1; j <= m; j++)
            fscanf(f,"%d",&A[i][j]);
    fclose(f);
}
```




```
8 12 5 6
1 1 0 0 0 0 1 0 0 0 0 1
1 0 0 0 1 1 0 1 0 0 1 1
0 0 1 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 1 0 0 1 0 1
1 0 0 1 0 0 0 0 0 1 0 0
1 0 1 0 1 0 0 0 1 0 1 0
0 0 0 0 1 0 1 0 0 0 0 0
1 0 1 1 0 1 1 1 0 1 0 1
```

MAZE problem

```
void init(){
    head = NULL; tail = NULL;
}
int isEmpty(){
    return head == NULL && tail == NULL;
}
void push(Node * node){
    if(isEmpty()){ head = node; tail = node;}
    else{ tail->next = node; tail = node;}
}
Node* pop(){
    if(isEmpty()) return NULL;
    Node* node = head;     head = node->next;
    if(head == NULL) tail = NULL;
    return node;
}
```

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST

MAZE problem

```
int main(){
    input();
    for(int r = 1; r <= n; r++)
        for(int c = 1; c <= m; c++)
            visited[r][c] = 0;
    init();
    Node* startNode = makeNode(r0,c0);
    Node* finalNode; //row < 1 || row > n || col < 1 || col > m
    // Do BFS(startNode):

      ............ .
       if (finalNode == NULL)
        printf("No solution out of the maze");
    else
       printf("Found solution out of the maze");
```

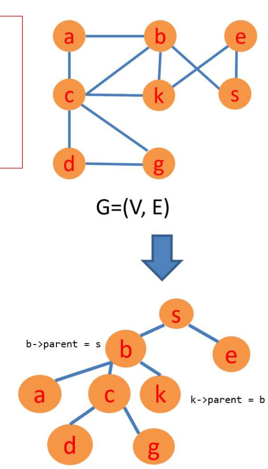
How can print on the screen the way to escape the maze ???

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

MAZE problem

```
typedef struct Node{
    int row,col;// the index of row and column in the maze of the node
    struct Node* next; // pointed to the next node in the queue
    struct Node* parent;
}Node;
```

```
Node* makeNode(int row, int col, Node *parent)
{
    Node* node = (Node*)malloc(sizeof(Node));
    node->row = row; node->col = col; node->next = NULL;
    node->parent = parent;
    return node;
}
```



NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST