HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

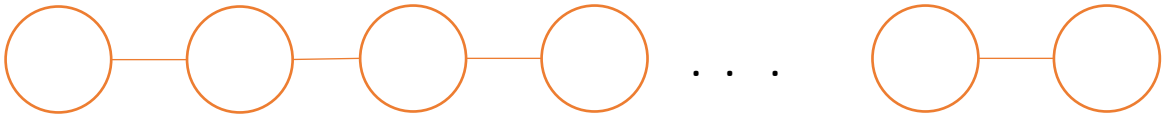
Lists

Content

- Definition
- Abstract Data Type
- Array
- Linked lists
- Stacks
- Queues

Definition

- Data structures store objects in a linear structure
- Specify the first object of the list
- Each object: has a unique successor



Abstract Data Type

- Notations:
  - *L*: a list
  - *x*: an object (element) of the list
  - *p*: position type
  - END(*L*): return the position after the position of the last element of *L*

Abstract Data Type

- Operations
  - Insert(x, p, L): insert an element x at the position p in L
  - Locate(x, L): return the position of x in L
  - Retrieve(p, L): return the element at the position p in L
  - Delete(p, L): remove the element at the position p in L
  - Next(p, L): return the next position of p in L
  - Prev(p, L): return the previous position of p in L
  - MakeNull(L): make L null
  - First(L): return the first position of L

Array

- Objects are allocated continuously
- Access objects via indices
- Insertions and removals need to consolidate elements

Array

- Declaration
  - **int a[100000];** // array
  - **int n;** // number of active
    // elements (started from 0)

```
void insert(int x, int p) {
  for(int j = n-1; j >= p; j--)
    a[j+1] = a[j];
  a[p] = x; n = n + 1;
}
void delete(int p) {
  for(int j = p+1; j <= n-1; j++)
    a[j-1] = a[j];
  n = n - 1;
}
int retrieve(int p) {
  return a[p];
}
```

Array

- Declaration
  - **int a[100000];** // array
  - **int n;** // number of active //elements (started from 0)

```
int locate(int x) {// vi trí đầu tiên của
x trong danh sách
  for(int j= 0; j <= n-1; j++)
    if(a[j] == x) return j;
  return -1;
}
void makeNull() {
  n = 0;
}
int next(int p) {
  if(0 <= p && p < n-1) return p+1;
  return -1;
}
int prev(int p) {
  if(p > 0 && p <= n-1) return p-1;
  return -1;
}
```

Pointers and linked lists

- Pointers: address of variables, objects in the memory
- **`int* p: p`** is a pointer to an **int** variable. Value of **p** specifies the address of the variable it points
- **`int a; int* p = &a;`**// p points to a
- Structure

```
typedef struct  TNode{
        int a;
        double b;
        char* s;
}TNode;
```

- **`TNode* q: q`** is a pointer to a variable of type **`TNode`**

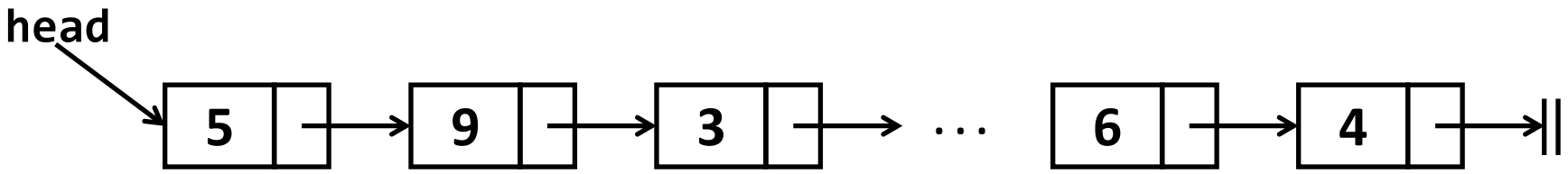
Pointers and linked lists

- **`q->a:`** access the the field **`a`** of the object
- **`q = (TNode*)malloc(sizeof(TNode)):`** allocate an object pointed by **`q`**

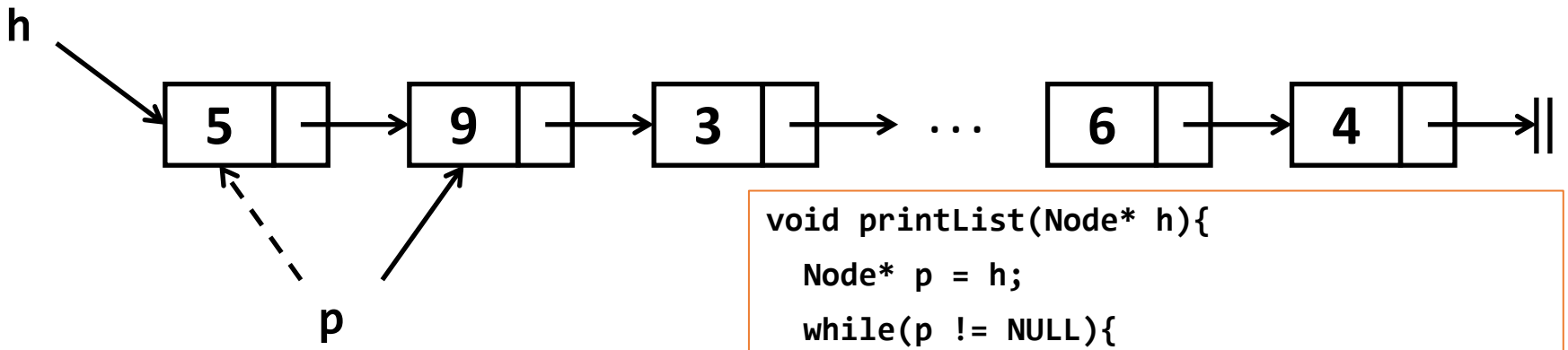
```
int main() {
    int a = 123;
    int* p;
    p = &a;
    *p = 456;
    printf("a = %d\n",a);
}
```

Single linked lists

```
struct Node{
    int value;
    Node* next; // pointer to the successor
};
Node* head; // pointer to the first element of the list
```

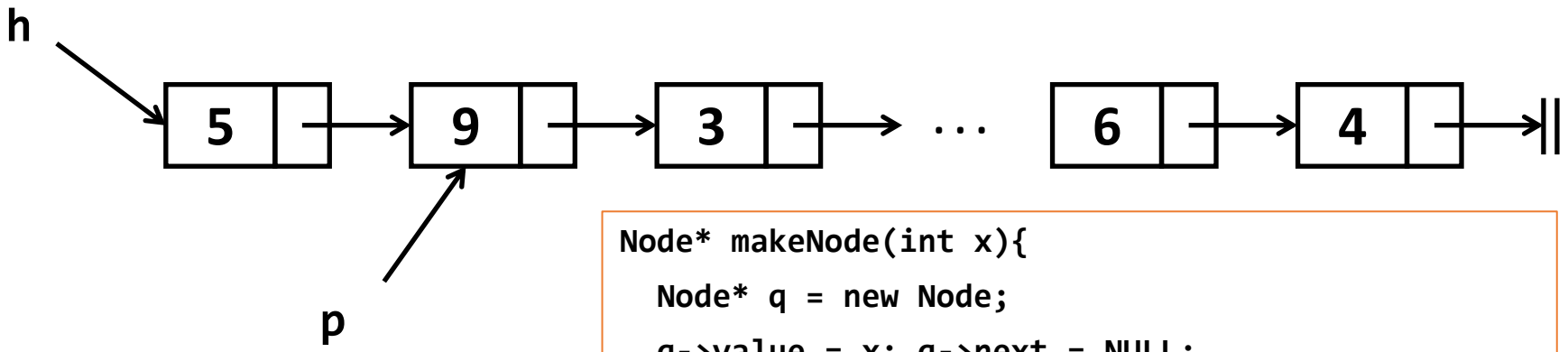


Single linked lists



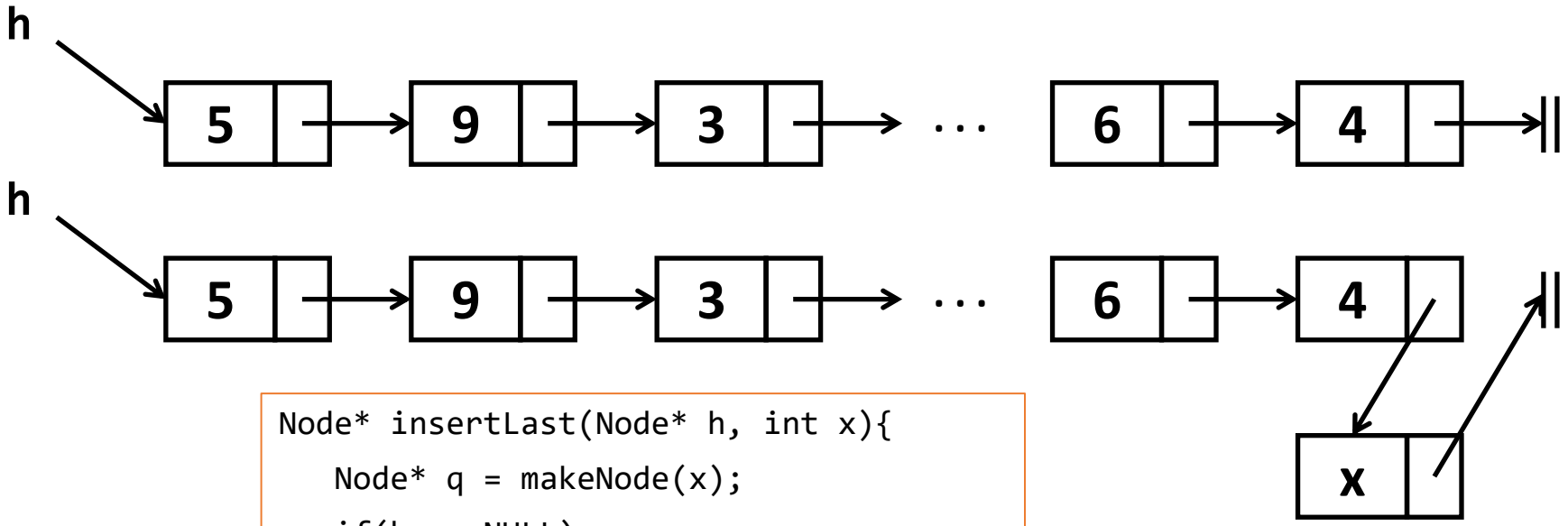

```
void printList(Node* h){
  Node* p = h;
  while(p != NULL){
    printf("%d ",p->value);
    p = p->next;
  }
}
Node* findLast(Node* h){
  Node* p = h;
  while(p != NULL){
    if(p->next == NULL) return p;
    p = p->next;
  }
  return NULL;
}
```

Single linked lists



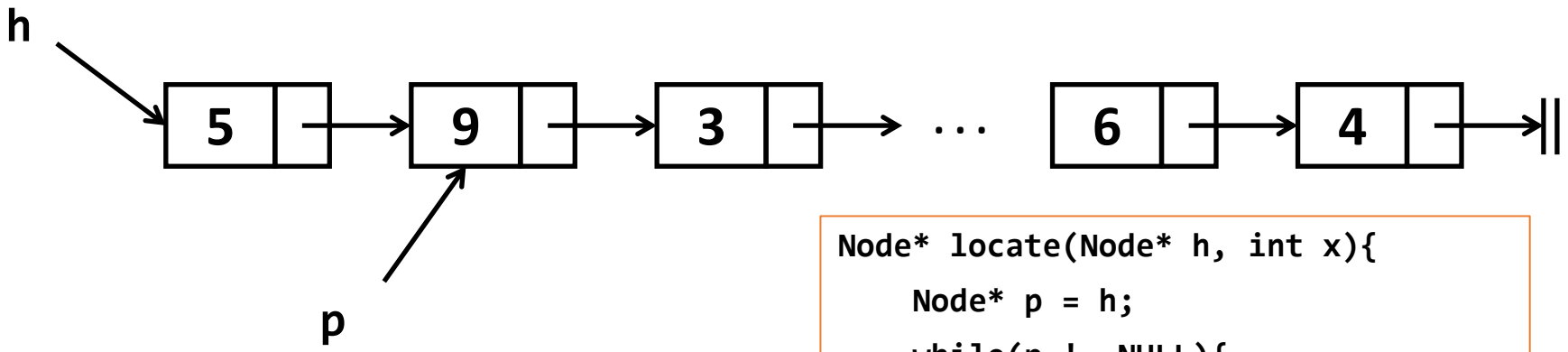

```
Node* makeNode(int x){
  Node* q = new Node;
  q->value = x; q->next = NULL;
  return q;
}

Node* insertAfter(Node* h, Node* p, int x){
  if(p == NULL) return h;
  Node* q = makeNode(x);
  if(h == NULL) return q;
  q->next = p->next;
  p->next = q;
  return h;
}
```

Single linked lists



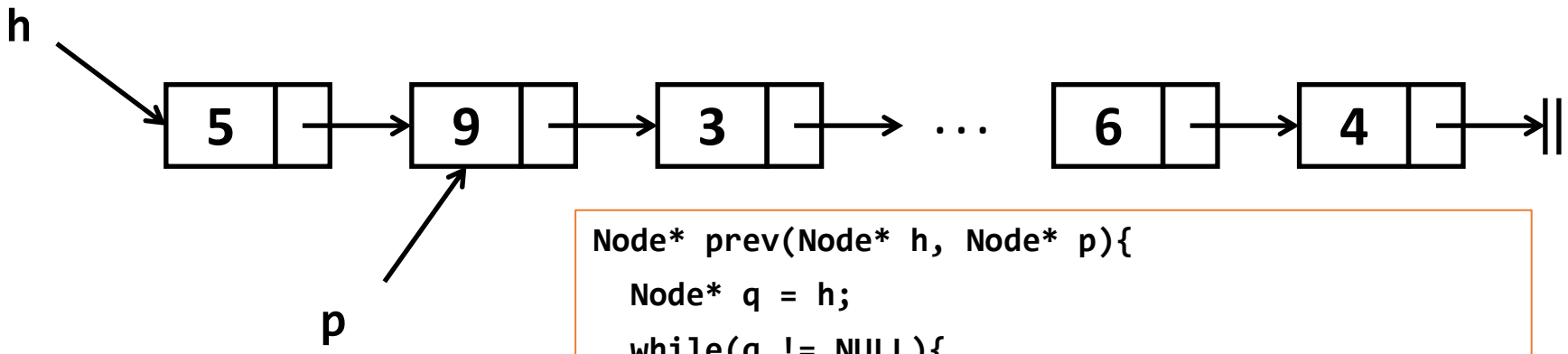

```
Node* insertLast(Node* h, int x){
   Node* q = makeNode(x);
   if(h == NULL)
     return q;
   Node* p = h;
   while(p->next != NULL)
     p = p->next;
   p->next = q;
   return h;
}
```

Single linked lists



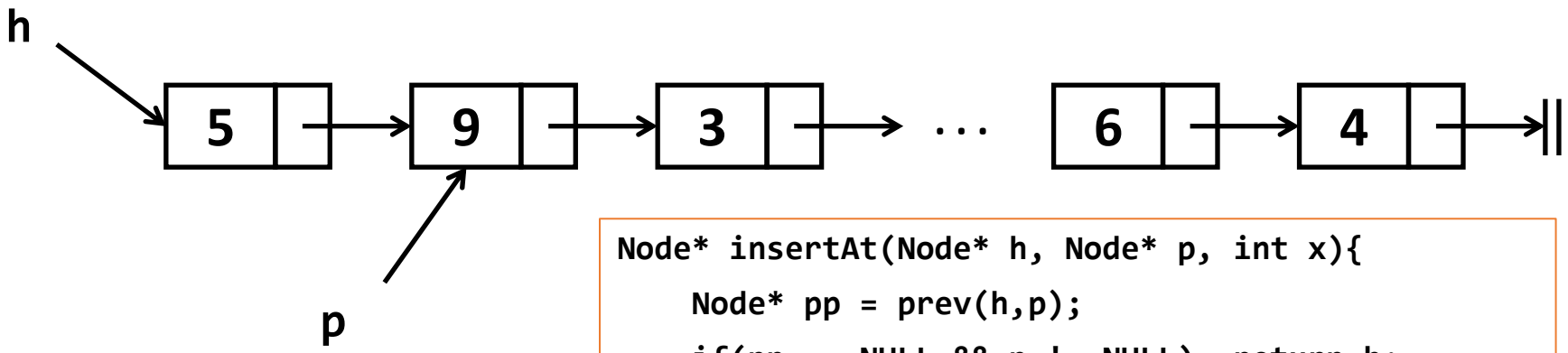

```
Node* locate(Node* h, int x){
    Node* p = h;
    while(p != NULL){
     if(p->value == x) return p;
        p = p->next;
    }
    return NULL;
}
```

Single linked lists



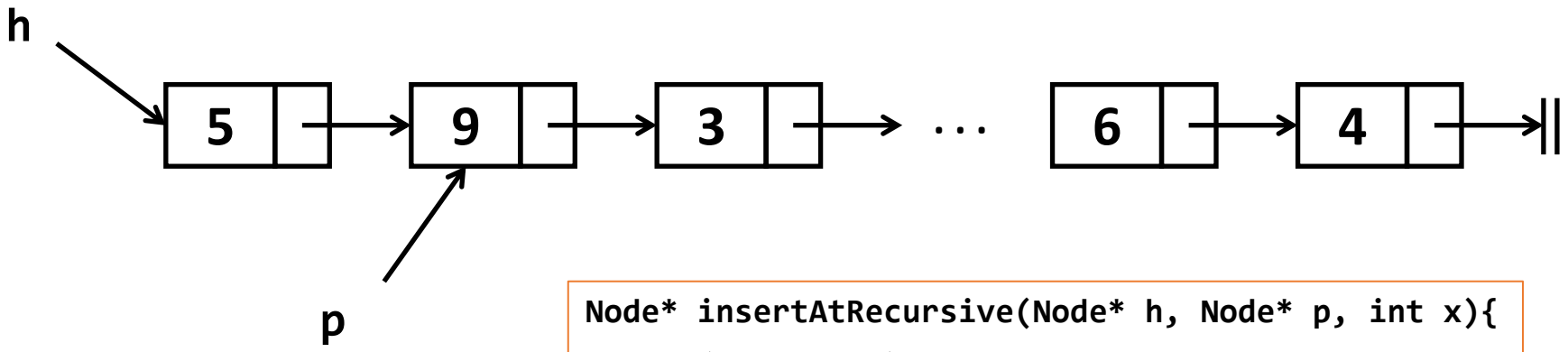

```
Node* prev(Node* h, Node* p){
  Node* q = h;
  while(q != NULL){
    if(q->next == p) return q;
    q = q->next;
  }
  return NULL;
}
```

Single linked lists



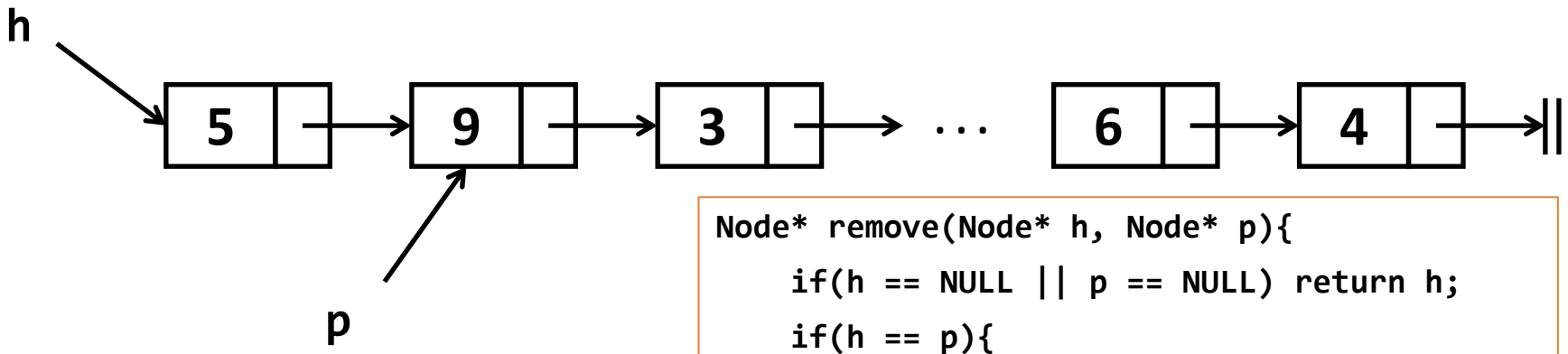

```
Node* insertAt(Node* h, Node* p, int x){
    Node* pp = prev(h,p);
    if(pp == NULL && p != NULL)  return h;
    Node* q = new Node;
    q->value = x; q->next = NULL;
    if(pp == NULL){
        if(h == NULL)
            return q;
        q->next = h;
        return q;
    }
    q->next = p;    pp->next = q;
    return h;
}
```

Single linked lists



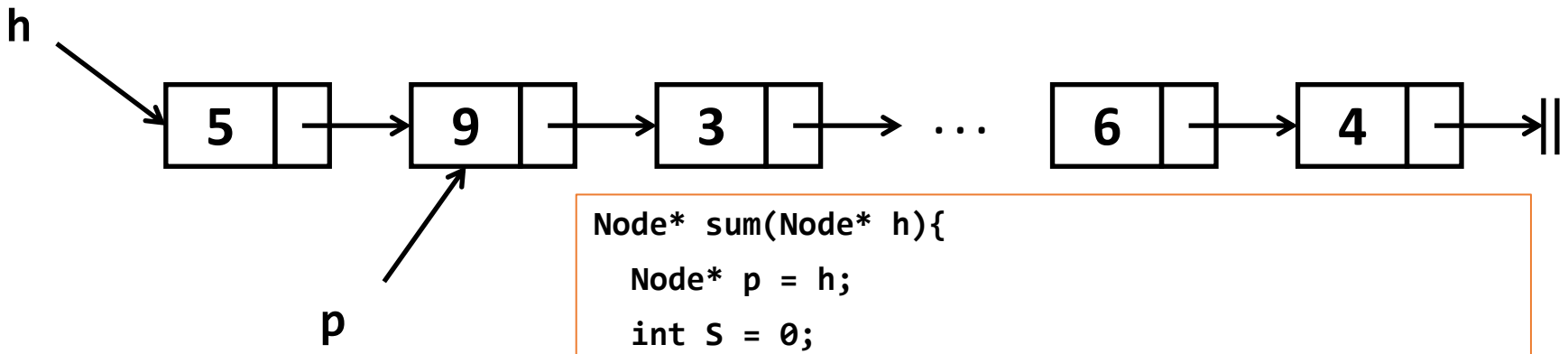

```
Node* insertAtRecursive(Node* h, Node* p, int x){
    if(p == NULL) return h;
    if(h == NULL || p == h){
      return makeNode(x);
    }else{
        h->next = insertAtRecursive(h->next,p,x);
        return h;
    }
}
```

Single linked lists




```
Node* remove(Node* h, Node* p){
    if(h == NULL || p == NULL) return h;
    if(h == p){
        h = h->next;
        delete p;
        return h;
    }else{
        h->next = remove(h->next,p);
        return h;
    }
}
```

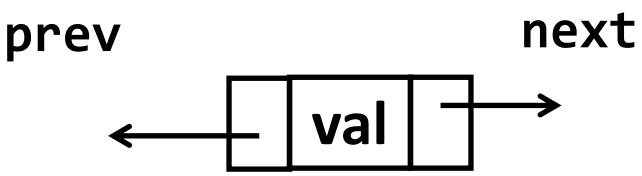
Single linked lists

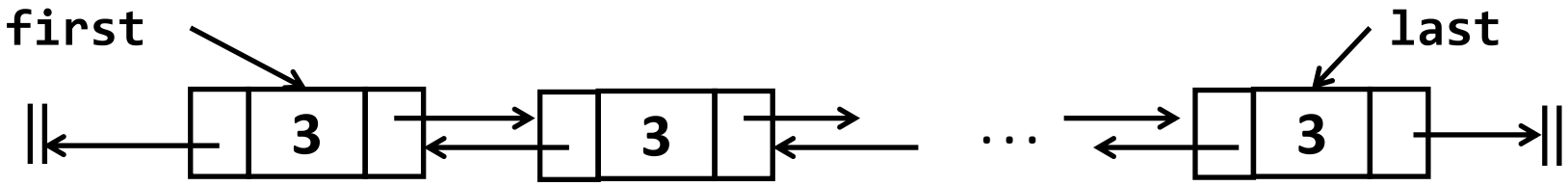



```
Node* sum(Node* h){
  Node* p = h;
  int S = 0;
  while(p != NULL) {
    S = S + p->value;
    p = p->next;
  }
  return S;
}
int sumRecursive(Node* h){
    if(h == NULL) return 0;
    return h->value + sumRecursive(h->next);
}
```

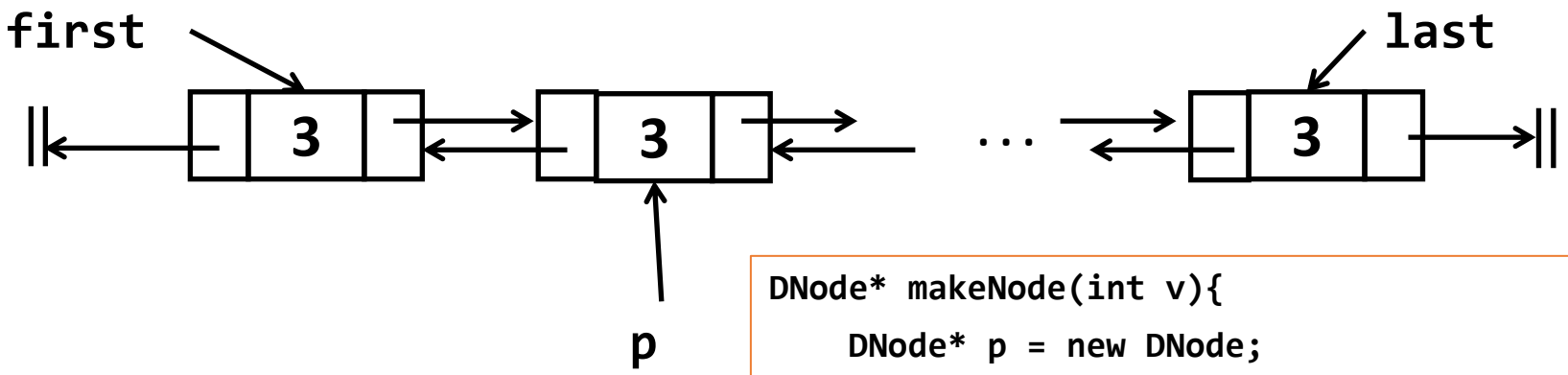
Doubly linked lists

```
struct DNode{
    int val;
    DNode* prev;// pointer to the predecessor
    DNode* next;// pointer to the successor
};

DNode* first;// pointer to the first element
DNode* last;// pointer to the last element
```



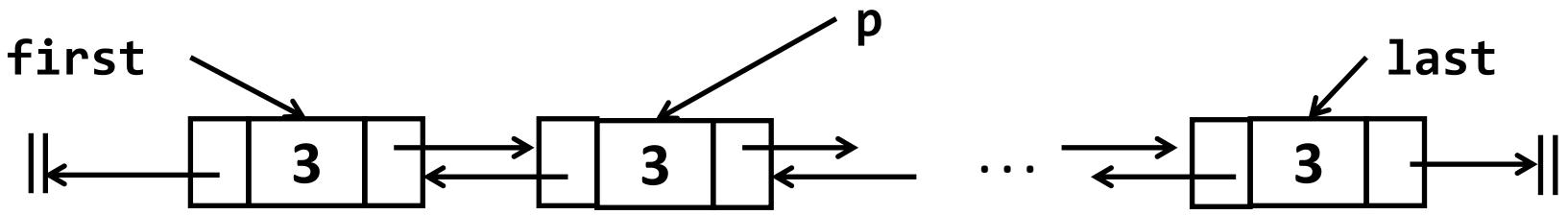


Doubly linked lists



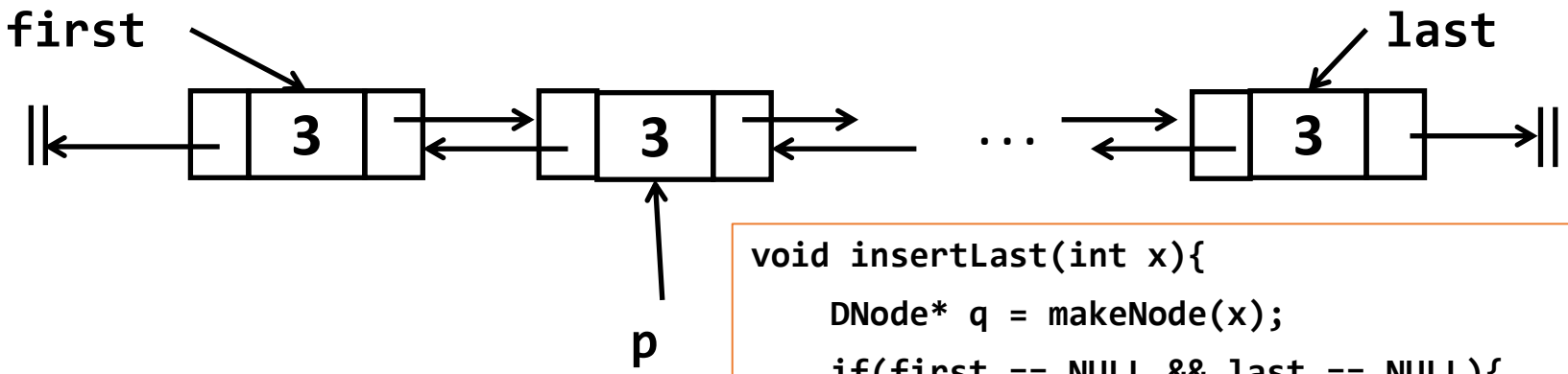

```
DNode* makeNode(int v){
    DNode* p = new DNode;
    p->val = v;
    p->next = NULL;
    p->prev = NULL;
    return p;
}
```

Doubly linked lists




```
void remove(DNode* p) {
  if(p == NULL) return;
  if(first == last && p == first){
    first = NULL; last = NULL; delete p;
  }
  if(p == first){
    first = p->next; first->prev = NULL;
    delete p; return;
  }
  if(p == last){
    last = p->prev; last->next = NULL;
    delete p; return;
  }
  p->prev->next = p->next;  p->next->prev = p->prev;  delete p;
}
```

Doubly linked lists




```
void insertLast(int x){
    DNode* q = makeNode(x);
    if(first == NULL && last == NULL){
        first = q;
        last = q;
        return;
    }
    q->prev = last;
    last->next = q;
    last = q;
}
```

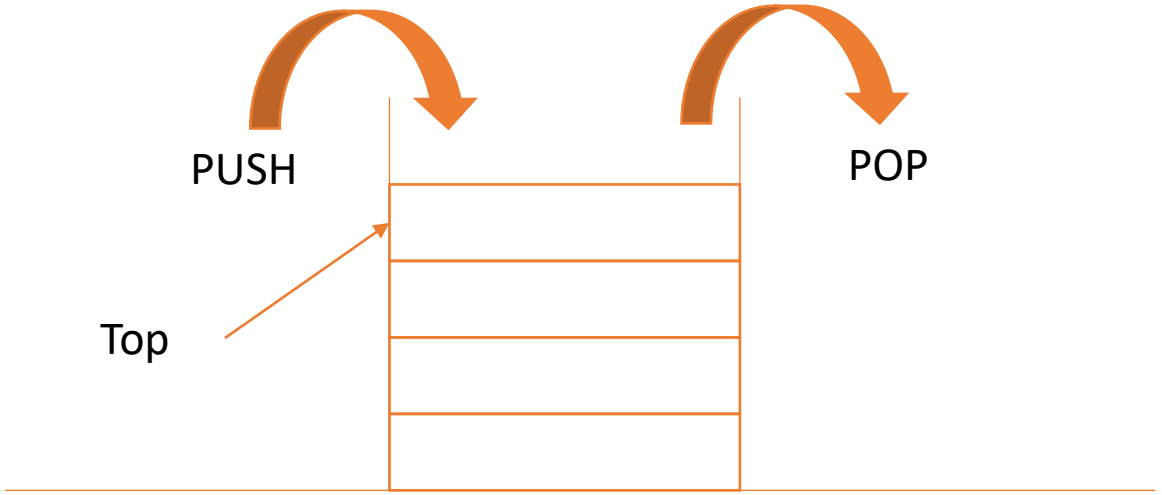
C++ library

- list: http://www.cplusplus.com/reference/list/list/
- operators
  - push_front()
  - push_back()
  - pop_front()
  - pop_back()
  - size()
  - clear()
  - erase()

```
#include <list>
#include <stdio.h>
using namespace std;

int main(){
  list<int> L;
  for(int i = 1; i <= 10; i++)
    L.push_back(i);
  list<int>::iterator it;
  for(it = L.begin(); it != L.end();
        it++){
     int x = *it;
     printf("%d ",x);
  }
}
```

Stacks

- Store elements in a linear structure
- Insertions and removals are performed at the top of the stack (Last In First Out – LIFO principle)
- Operations
  - Push(x, S): push an element x into the stack
  - Pop(S): remove an element out of the stack
  - Top(S): Access to the element at the top position of the stack
  - Empty(S): return true if the stack is empty



Stacks

- Application: Check the correctness of a parenthesis
  - [({})](): true
  - ([} {}): false

Stacks

- Application: Check the correctness of a parenthesis
  - [({})](): true
  - ([} {}): false
- Algorithm:
  - Initialize a stack S
  - Scan the parenthesis from left to right
    - Meet an opening parenthesis, then push it into S
    - Meet a closing parenthesis **A**,
      - If S is empty, then return FALSE
      - Otherwise, remove an opening parenthesis **B** out of S, if **B** and **A** do not match to each other, then return FALSE
    - Termination, if S is not empty, then return FALSE, otherwise, return TRUE

Stacks

- Parenthesis checking

```
#include <stack>
#include <stdio.h>

using namespace std;

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

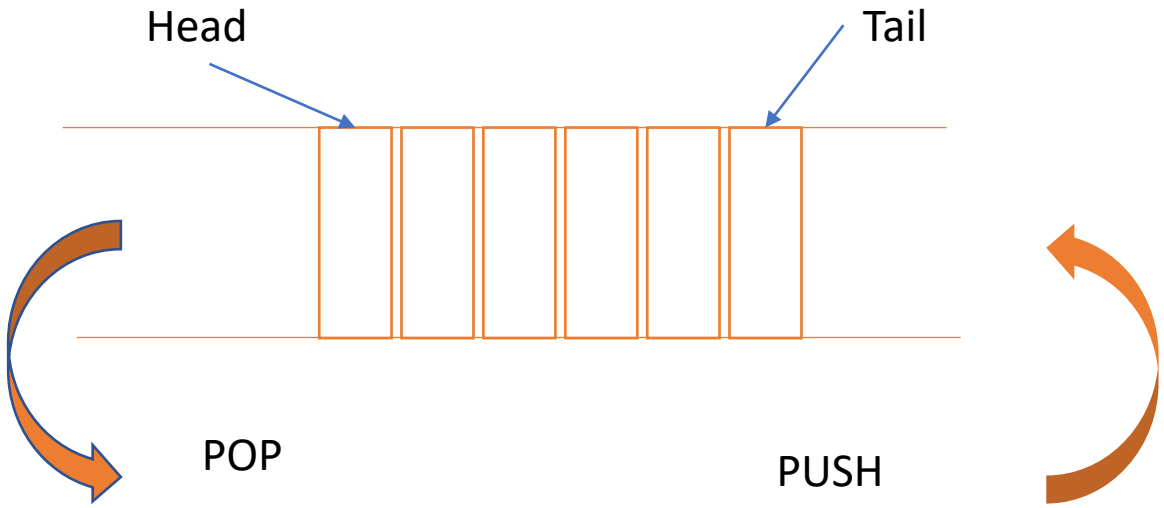
Stacks

- Parenthesis checking

```
bool solve(char* x, int n){
 stack<char> S;
 for(int i = 0; i <= n-1; i++){
   if(x[i] == '[' || x[i] == '(' || x[i] == '{'){
      S.push(x[i]);
   }else{
      if(S.empty()){
        return false;
      }else{
        char c = S.top(); S.pop();
        if(!match(c,x[i]))  return false;
      }
    }
  }
  return S.empty();
}
int main() {
  bool ok = solve("[({})]()",8);
}
```

Queues

- Lists with **head** and **tail**
- Insertions are performed at **tail**
- Removals are performed at **head** (First In First Out – FIFO)
- Operations
  - Enqueue(x, Q): insert an element x to the queue
  - Dequeue(Q): remove an element out of the queue
  - Empty(Q): return true if the queue is empty



Queues

- Application: Water jug
  - Given two jugs of capacities *a* and *b* (litres). Find to get *c* litres (a,b,c are positive integers) with following operations
    - Empty a jug
    - Fill a jug
    - Pour water from a jug to another jug

Queues

- Water jug: a = 6, b = 8, c = 4

Step	Operations	State
1	Fill the jug 1	(6,0)
2	Pour water from jug 1 to jug 2	(0,6)
3	Fill the jug 1	(6,6)
4	Pour water from jug 1 to jug 2	(4,8)

Queues

- Design data structures
  - State (x, y): amount of water in the jugs 1 and 2
  - Initial state (0, 0)
  - Target state: $x = c$ or $y = c$ or $x + y = c$
  - State transition
    - (1) Fill the jug 1: (a, y)
    - (2) Fill the jug 2: (x, b)
    - (3) Empty the jug 1: (0, y)
    - (4) Empty the jug: (x, 0)
    - (5) Pour water from jug 1 to jug 2: ($x + y - b$, b), if $x + y \geq b$
    - (6) Pour water from jug 1 to jug 2: (0, $x + y$), if $x + y \leq b$
    - (7) Pour water from jug 2 to jug 1: (a, $x + y - a$), if $x + y \geq a$
    - (8) Pour water from jug 2 to jug 1: ($x + y$, 0), if $x + y \leq a$

Queues

- Push state (0,0) into the queue

(0,0)									

- Pop state (0,0) out and push (6,0), (0,8) into the queue

(0,0)	(6,0)	(0,8)							

- Pop state (6,0) out and push (0,6) and (6,8) into the queue

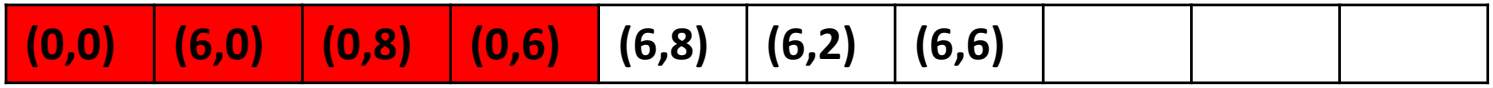
(0,0)	(6,0)	(0,8)	(0,6)	(6,8)					

- Pop state (0,8) out and push (6,2) into the queue

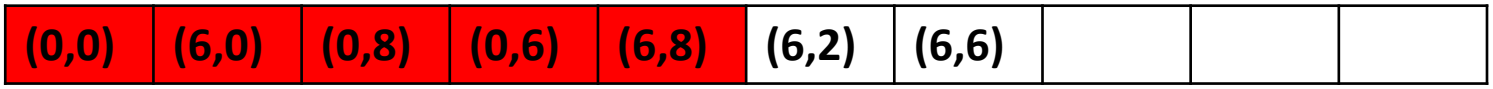
(0,0)	(6,0)	(0,8)	(0,6)	(6,8)	(6,2)				

Queues

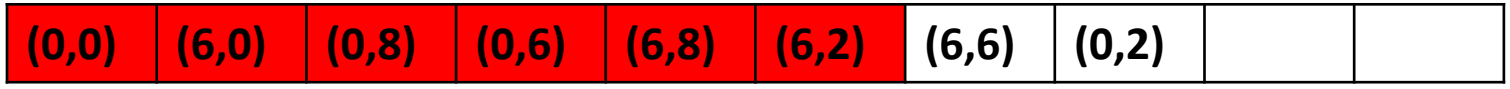
- Pop state (0,6) out and push (6,6) into the queue



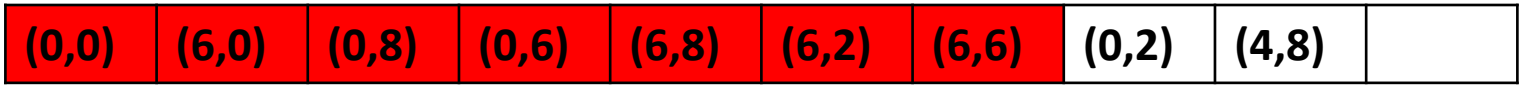
- Pop state (6,8) out, do not generate new states



- Pop state (6,2) out and push (0,2) into the queue



- Pop state (6,6) out and push (4,8) into the queue



Queues

- Design data structures
  - Initialize a queue Q for storing generated states
  - 2-dimensional array 2 for marking generated states
    - visited[*x*][*y*] = true, if the state (*x*, *y*) has been generated

Queues

- Data structures declaration

```
#include <stdio.h>
#include <stdlib.h>
#include <queue>
#include <stack>
#include <list>
using namespace std;
struct State{
  int x;
  int y;
  char* msg;// action to generate current state
  State* p;// pointer to the state generating current state
};
bool visited[10000][10000];
queue<State*> Q;
list<State*> L;
State* target;
int a,b,c;
```

Queues

- Initialize data structures

```
void initVisited(){
  for(int x = 0; x < 10000; x++)
    for(int y = 0; y < 10000; y++)
      visited[x][y] = false;
}
bool reachTarget(State* S){
    return S->x == c || S->y == c ||
       S->x + S->y == c;
}
void markVisit(State* S){
    visited[S->x][S->y] = true;
}
void freeMemory(){
    list<State*>::iterator it;
    for(it = L.begin(); it != L.end(); it++){
        delete *it;
    }
}
```

Queues

- Generate a new state by empty jug 1

```
bool genMove1Out(State* S){
    if(visited[0][S->y]) return false;
    State* newS = new State;
    newS->x = 0;
    newS->y = S->y;
    newS->msg = "Do het nuoc o coc 1 ra ngoai";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by empty jug 2

```
bool genMove2Out(State* S){
    if(visited[S->x][0]) return false;
    State* newS = new State;
    newS->x = S->x;
    newS->y = 0;
    newS->msg = "Do het nuoc o coc 2 ra ngoai";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by pouring water from jug 1 to jug 2 (case 1)

```
bool genMove1Full2(State* S){
    if(S->x+S->y < b) return false;
    if(visited[S->x + S->y - b][b]) return false;
    State* newS = new State;
    newS->x = S->x + S->y - b;
    newS->y = b;
    newS->msg = "Do nuoc tu coc 1 vao day coc 2";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by pouring water from jug 2 to jug 1 (case 1)

```
bool genMove2Full1(State* S){
    if(S->x+S->y < a) return false;
    if(visited[a][S->x + S->y - a]) return false;
    State* newS = new State;
    newS->x = a;
    newS->y = S->x + S->y - a;
    newS->msg = "Do nuoc tu coc 2 vao day coc 1";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by pouring water from jug 1 to jug 2 (case 2)

```
bool genMoveAll12(State* S){
    if(S->x + S->y > b) return false;
    if(visited[0][S->x + S->y]) return false;
    State* newS = new State;
    newS->x = 0;
    newS->y = S->x + S->y;
    newS->msg = "Do het nuoc tu coc 1 sang coc 2";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by pouring water from jug 2 to jug 1 (case 2)

```
bool genMoveAll21(State* S){
    if(S->x + S->y > a) return false;
    if(visited[S->x + S->y][0]) return false;
    State* newS = new State;
    newS->x = S->x + S->y;
    newS->y = 0;
    newS->msg = "Do het nuoc tu coc 2 sang coc 1";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by filling jug 1

```
bool genMoveFill1(State* S){
    if(visited[a][S->y]) return false;
    State* newS = new State;
    newS->x = a;
    newS->y = S->y;
    newS->msg = "Do day nuoc vao coc 1";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;
}
```

Queues

- Generate a new state by filling jug 2

```
bool genMoveFill2(State* S){
    if(visited[S->x][b]) return false;
    State* newS = new State;
    newS->x = S->x;
    newS->y = b;
    newS->msg = "Do day nuoc vao coc 2";
    newS->p = S;
    Q.push(newS); markVisit(newS);
    L.push_back(newS);
    if(reachTarget(newS)){
        target = newS;
        return true;
    }
    return false;

}
```

Queues

- Print the sequence of operations for obtaining the objective

```
void print(State* target){
    printf("----------RESULT-----------\n");
    if(target == NULL)
       printf("Khong co loi giai!!!!!!!");
    State* currentS = target;
    stack<State*> actions;
    while(currentS != NULL){
        actions.push(currentS);
        currentS = currentS->p;
    }
    while(actions.size() > 0){
        currentS = actions.top();
        actions.pop();
        printf("%s, (%d,%d)\n",
             currentS->msg,currentS->x,
                 currentS->y);
    }
}
```

Queues

- Main process for generating states and push them into the queue

```
void solve(){
  initVisited();
  // sinh ra trang thai ban dau (0,0) va dua vao Q
  State* S = new State;
  S->x = 0; S->y = 0; S->p = NULL;
  Q.push(S); markVisit(S);
  while(!Q.empty()){
    State* S = Q.front(); Q.pop();
    if(genMove1Out(S)) break;
    if(genMove2Out(S)) break;
    if(genMove1Full2(S)) break;
    if(genMoveAll12(S)) break;
    if(genMove2Full1(S)) break;
    if(genMoveAll21(S)) break;
    if(genMoveFill1(S)) break;
    if(genMoveFill2(S)) break;
  }
}
```

Queues

- Example with: a = 4, b = 7, c = 9

```
int main(){
    a = 4;
    b = 7;
    c = 9;
    target = NULL;
    solve();
    print(target);
    freeMemory();
}
```