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

Chapter 1. Basic data types, I/O with files

Chapter 2. Recursion

Chapter 3. Lists

Chapter 4. Stack and Queue

Chapter 5. Trees

Chapter 6. Sorting

Chapter 7. Searching

2

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

Chapter 3. Lists

Nguyễn Khánh Phương

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

Contents

1. Basic operations on linked-list
2. Stack
3. Queue

Contents

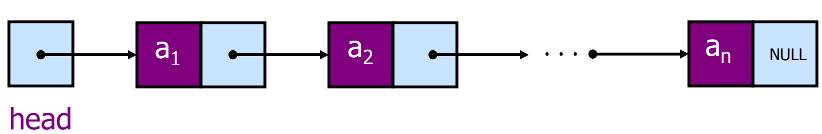
1. **Basic operations on linked-list**
2. Stack
3. Queue

Basic operations on linked lists

- Each node of a linked is defined as follows

```
typedef struct Node{
      int data;
      struct Node* next;
}Node;
```

- Implement following operations
  - `Node* insertLast(Node* head, int v);`// insert a node at that last position
  - `Node* removeNode(Node* head, int v);`// remove a first node having value v
  - `Node* removeAll(Node* head, int v);`// remove all nodes having value v
  - `int countNodes(Node* head);`// count number of nodes
  - `Node* reverse(Node* head);`// reverse the linked list



Basic operations on linked lists

- Define data structures:

```
#include <stdio.h>
typedef struct Node{
    int data;
    struct Node* next; //point to the next element of the current element
}Node;

Node *makeNode(int v)  //allocate memory for a new node
{
    Node* p = (Node*)malloc(sizeof(Node));
    p->data = v; p->next = NULL;
    return p;
}
```

7

Basic operations on linked lists

- Insert a node to the end of a linked list (use and not use recursion)

```
Node* insertLast(Node* head, int v)
{
   if (head == NULL)
            return makeNode(v);

    //Move to the end:
    Node* last = head;
    while(last->next != NULL)
            last = last->next;

    Node* new_node = makeNode(v);
    last->next = new_node;
    return head;
}
```

```
Node* insertLastRecursive(Node* head, int v)
{
    if (head == NULL)
        return makeNode(v);

    head->next =
      insertLastRecursive(head->next, v);
    return head;
}
```

8

Basic operations on linked lists

- Insert a node to the end of a linked list (use and not use recursion)

```
Node* insertLast(Node* head, int v)
{
   if (head == NULL)
            return makeNode(v);

    //Move to the end:
    Node* last = head;
    while(last->next != NULL)
            last = last->next;

    Node* new_node = makeNode(v);
    last->next = new_node;
    return head;
}
```

```
void insertLast(Node** head_ref, int v)
{
  //1. Create a new element new_node:
  Node* new_node = makeNode(v);
  if (*head_ref == NULL) //no elements in list
     *head_ref = new_node;
  else {
     //move to the end:
     Node *last = *head_ref;
     while (last->next != NULL)
            last = last->next;
     /*update the next pointer of the last node: */
     last->next = new_node;
  }
}
```

NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

Basic operations on linked lists

- Remove a first node having value v (not use recursion)

```
Node* removeNode(Node* head, int v){
    if (head == NULL) return NULL;
    if (head->data == v) {
        Node* tmp = head; head = head->next;
        free(tmp);  return head;
    }
    Node* p = head;
    while(p->next != NULL){
        if (p->next->data == v) break;
        p = p->next;
    }
    if(p->next != NULL){
        Node* q = p->next; p->next = q->next; free(q);
    }
    return head;
}
```

Basic operations on linked lists

- Remove a first node having value v (use recursion)

```
Node* removeNodeRecursive(Node* head, int v)
{
    if (head == NULL) return NULL;
    if (head->data == v)
    {
        Node* tmp = head; head = head->next; free(tmp); return head;
    }
    head->next = removeNodeRecursive(h->next, v);
    return head;
}
```

11

Basic operations on linked lists

- Remove all nodes having value v (use recursion)

```
Node* removeAll(Node* head, int v)
{
    // remove all nodes having value v from the linked list headed by head
    if(head == NULL) return NULL;
    if(head->data == v){
        Node* tmp = head; head = head->next; free(tmp);
        head = removeAll(head,v); // continue to remove other elements having value v
        return head;
    }
    head->next = removeAll(head->next,v);
    return head;
}
```

12

Basic operations on linked lists

- Count number of nodes in the linked list (use and not use recursion)

```
int countNodes(Node* head)
{
    int cnt = 0;
    Node* p = head;
    while(p != NULL){
        cnt += 1;
        p = p->next;
    }
    return cnt;
}
```

```
int countNodesRecursive(Node* head)
{
    if(head == NULL) return 0;
    return 1+countNodesRecursive(head->next);
}
```

13

Basic operations on linked lists

- Reverse a linked list

```
Node* reverse(Node *head)
{
    Node* p = head;
    Node* pp = NULL;
    Node* np = NULL;
    while(p != NULL){
        np = p->next;
        p->next = pp;
        pp = p;
        p = np;
    }
    return pp;
}
```

14

Exercise 1

- Input integer numbers from the keyboard, finish when user presses "enter". Store these numbers in a linked list

```
typedef struct Node{
    int data;
    struct Node *next;
}Node;
Node *head;
```

```
struct Node {
   int data;
   struct Node *next;
};
struct Node *head;
```

Require: Each time user enter a number, insert this number at

1. The beginning of the list

```
void push(Node** head_ref, int new_data);
```

```
Nhap so thu  1 = 20
Nhap so thu  2 = 19
Nhap so thu  3 = 30
Nhap so thu  4 = 15
Nhap so thu  5 =

 Danh sach lien ket chua 4 cac so nguyen vua nhap vao la:
  15  30  19  20
```

2. The end of the list

```
void append(Node** head_ref, int new_data);
```

```
Nhap so thu  1 = 20
Nhap so thu  2 = 19
Nhap so thu  3 = 30
Nhap so thu  4 = 15
Nhap so thu  5 =

 Danh sach lien ket chua 4 cac so nguyen vua nhap vao la:
  20  19  30  15
```

15

```c
#include<stdio.h>

typedef struct Node
{
  int data;
  struct Node *next;
}Node;

void push(Node** head_ref, int new_data);
void append(Node** head_ref, int new_data);
void printList(Node *node);

int main()
{
  Node* head = NULL; //Danh sach luc dau khong co phan tu nao
  int n = 0,data;
  int stop = 0;
  char line[32];
  do{
      printf("Nhap so thu  %d = ",n+1);
      fgets(line, sizeof(line), stdin);
      if (*line=='\n') stop = 1;
      else
      {
        data = atoi(line);
        push(&head,data);
         n++;
      }
  }while (stop == 0);

  printf("\n Danh sach lien ket chua %d cac so nguyen vua nhap vao la: \n ",n);
  printList(head);
  return 0;
}
```

16

```c
/*Print values of elements in the linked list starting
from the element pointed by the pointer node*/
void printList(Node *node)
{
  Node *cur;
  for (cur = node; cur != NULL; cur = cur->next)
      printf(" %d ", cur->data);
}

void printList1(Node *node)
{
  while (node != NULL){
     printf(" %d ", node->data);
     node = node->next;
  }
}
```

17

```c
/* Given a reference (pointer to pointer) to the head of a list
and an int,  inserts a new node on the front of the list. */
void push(Node** head_ref, int new_data){
    //1. Create a new element new_node:
    Node* new_node = (Node* ) malloc(sizeof(Node));
    new_node->data  = new_data;

    //2. Assign next of new_node to head:
    new_node->next = (*head_ref);

    //3. Move head to new_node:
    (*head_ref) = new_node;
}
```

18

```
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(Node** head_ref, int new_data)
{
  //1. Create a new element new_node:
  Node* new_node = (Node*) malloc(sizeof(Node));
  new_node->data  = new_data;
  //new_node is the last element in the list --> assign its next = NULL
  new_node->next = NULL;
  if (*head_ref == NULL) //list does not have any elements
     *head_ref = new_node;
  else {
       //move to the last position in the linked list:
       Node *last = *head_ref;
       while (last->next != NULL) last = last->next;
       //update the next pointer of the last node:
       last->next = new_node;
  }
}
```


Exercise 1 (cont.)

Insert 1 node:

1. At the beginning of the list

```
Node* push1(Node* head, int new_data)
void push(Node** head_ref, int new_data)
```

2. At the end of the list

```
Node* append1(Node* head, int new_data)
void append(Node** head_ref, int new_data)
```

3. After the position pointed by pointer cur

```
void insertAfter(Node* cur, int new_data)
```

4. Before the position pointed by pointer cur

```
void insertBefore(Node* cur, int new_data)
```

5. After the *kth* element in the list

```
Node* insertAfterItemK1(Node* head, int k, int new_data)
void insertAfterItemK(Node** head_ref, int k, int new_data)
```

Exercise 1 (cont.)

Delete 1 node:

1. At position pointed by pointer del (begin/last/middle of the list)

```
void deleteNode(Node **head_ref, Node *del);
```

2. At position *k*

```
void deleteNodeAtPosition(Node **head_ref, int k)
```

3. Have value = key (the first node in the list with value = key)

```
void deleteNodeKey(Node **head_ref, int key)
```


Insert 1 node at the beginning of the list

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int,  inserts a new node on the front of the list. */
void push(Node** head_ref, int new_data){
    //1. Tao them phan tu new_node:
    Node* new_node = (Node* ) malloc(sizeof(Node));
    new_node->data  = new_data;

    //2. Gan next cua new_node tro toi head:
    new_node->next = (*head_ref);

    //3. Di chuyen head tro toi new_node:
    (*head_ref) = new_node;
}


Node* push1(Node* head, int new_data){
    //1. Tao them phan tu new_node:
    Node* new_node = (Node* ) malloc(sizeof(Node));
    new_node->data  = new_data;

    //2. Gan next cua new_node tro toi head:
    new_node->next = head;

    //3. Di chuyen head tro toi new_node:
    head = new_node;
    return head;
}
```

Insert 1 node at the end of the list

```c
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(Node** head_ref, int new_data)
{
    //1. Tao them phan tu new_node:
    Node* new_node = (Node*) malloc(sizeof(Node));
    new_node->data  = new_data;
    new_node->next = NULL; //new_node se la node cuoi cung trong danh sach--> gan next = NULL

    if (*head_ref == NULL) //danh sach chua co phan tu nao
        *head_ref = new_node;
    else
    {
        //di chuyen den cuoi danh sach:
        Node *last = *head_ref;
        while (last->next != NULL) last = last->next;
        //thay doi con tro next cua node cuoi cung
        last->next = new_node;
    }
}
```

```c
Node* append1(Node* head, int new_data)
{
    //1. Tao them phan tu new_node:
    Node* new_node = (Node*) malloc(sizeof(Node));
    new_node->data  = new_data;
    new_node->next = NULL; //new_node se la node cuoi cung trong danh sach--> gan next = NULL

    if (head == NULL) //danh sach chua co phan tu nao
        head = new_node;
    else
    {
        //di chuyen den cuoi danh sach:
        Node *last = *head_ref;
        while (last->next != NULL) last = last->next;
        //thay doi con tro next cua node cuoi cung
        last->next = new_node;
    }
    return head;
}
```

23

Delete a node at position k

```c
void deleteNodeAtPosition(Node **head_ref, int k)
{
   // If linked list is empty
   if (*head_ref == NULL) return;

   // Store head node
   Node* temp = *head_ref;

    if (k == 1) // If head needs to be removed
    {
        *head_ref = temp->next;   // Change head
        free(temp);               // free old head
        return;
    }

    //1. Xac dinh temp tro toi node thu (k-1) (node ngay truoc node can xoa)
    for (int i=1; temp!=NULL && i<k-1; i++) temp = temp->next;

    // Neu danh sach co it hon k node
    if (temp == NULL || temp->next == NULL) return;

    //2. Xac dinh node temp1 tro toi node thu (k+1) (node ngay sau node can xoa)
    Node *temp1 = temp->next->next;

    //3. Xoa node thu k
    free(temp->next);  // Free memory

    //4. Cap nhat lai ket noi
    temp->next = temp1;
}
```

```c
Node* deleteNodeAtPosition1(Node *head, int k)
{
   // If linked list is empty
   if (head == NULL) return head;
   // Store head node
   Node* temp = head;

    if (k == 1) // If head needs to be removed
    {
        head = temp->next;   // Change head
        free(temp);               // free old head
        return head;
    }
    //1. Xac dinh temp tro toi node thu (k-1) (node ngay truoc node can xoa)
    for (int i=1; temp!=NULL && i<k-1; i++) temp = temp->next;
    // Neu danh sach co it hon k node
    if (temp == NULL || temp->next == NULL) return head;
    //2. Xac dinh node temp1 tro toi node thu (k+1) (node ngay sau node can xoa)
    Node *temp1 = temp->next->next;
    //3. Xoa node thu k
    free(temp->next);  // Free memory
    //4. Cap nhat lai ket noi
    temp->next = temp1;
    return head;
}
```

```c
/* Given a reference (pointer to pointer) to the head of a list and a key, deletes the first occurrence of key in linked list */
void deleteNodeKey(Node **head_ref, int key)
{
    // Store head node
    Node* temp = *head_ref, *prev;
    // Neu key xuat hien o node head
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;   // Changed head
        free(temp);               // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    //Neu trong danh sach khong co gia tri key:
    if (temp == NULL) {
        printf("Danh sach khong co gia tri %d\n",key);
        return;
    }

    //1. Cap nhat ket noi
    prev->next = temp->next;
    //2. Xoa node co gia tri key
    free(temp);  // Free memory
}
```

Delete a node with value = key

```c
Node* deleteNodeKey1(Node *head, int key)
{
    // Store head node
    Node* temp = head, *prev;
    // Neu key xuat hien o node head
    if (temp != NULL && temp->data == key) {
        head = temp->next;   // Changed head
        free(temp);               // free old head
        return head;
    }
    // Search for the key to be deleted, keep track of the previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    //Neu trong danh sach khong co gia tri key:
    if (temp == NULL) {
        printf("Danh sach khong co gia tri %d\n",key);
        return head;
    }
    //1. Cap nhat ket noi
    prev->next = temp->next;
    //2. Xoa node co gia tri key
    free(temp);  // Free memory
    return head;
}
```

Exercise 3: Profile management problem

```
List of Commands: Load Print Find Insert Remove Store Quit
Enter command: Insert
Enter studentID, email, grade: 2011234 phuong@gmail.com 7.5
List of Commands: Load Print Find Insert Remove Store Quit
Enter command: Insert
Enter studentID, email, grade: 2011342 quanganh@yahoo.com 8
List of Commands: Load Print Find Insert Remove Store Quit
Enter command: Print
2011234: phuong@gmail.com, 7.500000
2011342: quanganh@yahoo.com, 8.000000
List of Commands: Load Print Find Insert Remove Store Quit
Enter command:
```

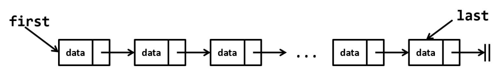
- A student profile consists of
  - id: ID of student
  - email: email of the student
  - grade: grade of Math course
- Write a program running in an interactive mode
  - Add a new profile at the end of the list
  - Load data from a text file into memory establishing a list
  - Print the information of students in the list
  - Find a profile with given id
  - Remove a profile with given id of student
  - Store the list into an external text file

26

Exercise 3: Profile management problem

- A student profile consists of
  - id: id of the student
  - email: email of the student
  - grade: grade of Math course

```
#include <stdio.h>

typedef struct Profile{
    char id[20];
    char email[100];
    double grade;
    struct Profile* next;
}Profile;

Profile* first, *last;
```



Exercise 3: Profile management problem

- Create a new node:
  - Allocate memory for the newNode;
  - Copy data to the newNode
  - newNode->next=NULL;

```
#include <stdlib.h>
Profile* makeProfile(char *id, char* email, double grade)
{
    Profile* newNode = (Profile*) malloc(sizeof(Profile));
    strcpy(newNode->id, id);
    strcpy(newNode->email,email);
    newNode->grade = grade;
    newNode->next = NULL;
    return newNode;
}
```

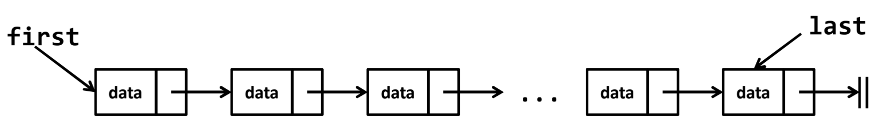
Exercise 3: Profile management problem

- A student profile consists of
  - id: ID of student
  - email: email of the student
  - grade: grade of Math course
- Write a program running in an interactive mode
  - **Add a new profile at the end of the list**
  - Load data from a text file into memory establishing a list
  - Print the information of students in the list
  - Find a profile with given id
  - Remove a profile with given id of student
  - Store the list into an external text file

29

Exercise 3: Profile management problem

- Add a new profile at the end of the list




```
void insertLast(char *id, char* email, double grade)
{
    Profile* profile = makeProfile(id,email,grade);
    if (first == NULL)
    {
         first = profile; last = profile;
    }
    else
    {
        last->next = profile; last = profile;
    }
}
```

30

Exercise 3: Profile management problem

- Write a program running in an interactive mode
  - **Load data from a text file into memory establishing a list**

profile - Notepad

File Edit Format View Help

```
20205140        anh.dqv205140@sis.hust.edu.vn   3.5
20205142        anh.mh205142@sis.hust.edu.vn    4
20176684        anh.nq176684@sis.hust.edu.vn    5.5
20205176        cong.bht205176@sis.hust.edu.vn  6
20205178        dat.nt205178@sis.hust.edu.vn    8
20194744        dung.ht194744@sis.hust.edu.vn   8.5
20200208        hien.bd200208@sis.hust.edu.vn   3
20205152        hoang.hh205152@sis.hust.edu.vn  2
20205154        hung.dt205154@sis.hust.edu.vn   6.5
20205155        hung.lt205155@sis.hust.edu.vn   8
20205157        hung.nv205157@sis.hust.edu.vn   7.5
20200270        huy.dn200270@sis.hust.edu.vn    9
20205159        linh.vt205159@sis.hust.edu.vn   10
20205161        long.pt205161@sis.hust.edu.vn   8.5
20205162        manh.dd205162@sis.hust.edu.vn   5
20200508        quan.nm200508@sis.hust.edu.vn   5.5
20205167        quang.nn205167@sis.hust.edu.vn  7
20205193        quy.vx205193@sis.hust.edu.vn    6.5
20205168        son.nh205168@sis.hust.edu.vn    4
20200597        thanh.vc200597@sis.hust.edu.vn  8
20205195        tien.nv205195@sis.hust.edu.vn   5.5
20194859        toan.pt194859@sis.hust.edu.vn   10
20200640        trung.hd200640@sis.hust.edu.vn  4.5
```


Exercise 3: Profile management problem

- **Load data from a text file into memory establishing a list**

```c
void loadFile()
{
    char filename[50];
    printf("Enter the name of file: ");scanf("%s",filename);
    FILE* f = fopen(filename,"r");
    if(f == NULL) printf("Load data -> file not found\n");

    while(!feof(f))
    {
        char id[20], email[100]; double grade;
        fscanf(f,"%s %s %lf",id, email, &grade);
        insertLast(id,email,grade);
    }
    fclose(f);
}
```

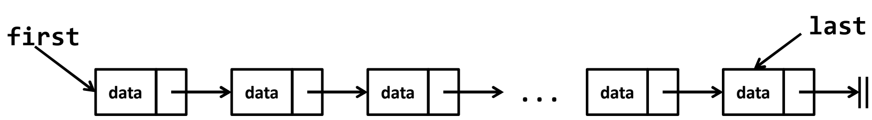
Exercise 3: Profile management problem

- A student profile consists of
  - id: ID of student
  - email: email of the student
  - grade: grade of Math course
- Write a program running in an interactive mode
  - Add a new profile at the end of the list
  - Load data from a text file into memory establishing a list
  - **Print the information of students in the list**
  - Find a profile with given id
  - Remove a profile with given id of student
  - Store the list into an external text file

33

Exercise 3: Profile management problem

- Print the information of students




```
void printList()
{
    Profile* p = first;
    while (p != NULL)
    {
        printf("%s: %s, %lf\n",p->id, p->email, p->grade);
        p = p->next;
    }
}
```

```
void printList()
{
    for (Profile* p = first; p != NULL; p = p->next)
        printf("%s: %s, %lf\n",p->id, p->email, p->grade);
}
```

34

Exercise 3: Profile management problem

- A student profile consists of
  - id: ID of student
  - email: email of the student
  - grade: grade of Math course
- Write a program running in an interactive mode
  - Add a new profile at the end of the list
  - Load data from a text file into memory establishing a list
  - Print the information of students in the list
  - **Find a profile with given id**
  - Remove a profile with given id of student
  - Store the list into an external text file

35

Exercise 3: Profile management problem

- Find a profile with given id

```
void findProfile()
{
    char id[20];
    printf("Enter the studentID that you want to find: ");scanf("%s",id);
    Profile* found = NULL;
    for(Profile* cur = first; cur != NULL; cur = cur->next)
    {
        if(strcmp(cur->id,id)==0)
        {
            found = cur; break;
        }
    }
    if(found == NULL)
       printf("NOT FOUND profile with id %s\n",id);
    else
       printf("FOUND profile id = %s has email = %s, grade = %lf\n",found->id,found->email, found->grade);
}
```

36

Exercise 3: Profile management problem

– Remove a profile with given id

```
void removeProfile() {
    if (first == NULL) { printf("The list is empty!!! ");return;}
    char id[20];
    printf("Enter the studentID that you want to find: ");scanf("%s",id);
    Profile *cur = first, *prev = NULL;
    while (cur != NULL)
    {
          if (strcmp(cur->id,id) == 0) //delete node cur
          {
             if (prev == NULL) //delete cur which is the first node:
             {
                first = cur->next;
                if (cur == last) last = first; //after deletion, the list consists only 1 node

             }
             else //cur is not the first node:
             {
                prev->next = cur->next;
                if (cur == last) last = prev;//cur is the last node
             }
             free(cur);
          }
          prev = cur; cur = cur->next;
    }
}
```

37

Exercise 3: Profile management problem

- A student profile consists of
  – id: ID of student
  – email: email of the student
  – grade: grade of Math course
- Write a program running in an interactive mode
  – Add a new profile at the end of the list
  – Load data from a text file into memory establishing a list
  – Print the information of students in the list
  – Find a profile with given id
  – Remove a profile with given id of student
  – **Store the list into an external text file**

NGUYỄN KHÁNH PHƯƠNG
SOICT– HUST 38

Exercise 3: Profile management problem

– Store the list into an external text file

```
void storeFile()
{
  char filename[256];
  printf("Enter the name of file: ");scanf("%s",filename);
  FILE* f = fopen(filename,"w");
  for(Profile* temp = first; temp != NULL; temp = temp->next)
  {
        fprintf(f,"%s %s %lf",temp->id,temp->email,temp->grade);
        if (temp->next != NULL) fprintf(f,"\n");
  }
  fclose(f);
}
```


Exercise 3: Profile management problem

```
void FREE_MEM()
{
    Profile* del = first;
    while (del != NULL)
    {
        Profile* first = first->next;
        free(del);
        del = first;
    }
}
```

Exercise 3: Profile management problem

```
void insert() {
    char id[20], email[100]; double grade;
    printf("Enter studentID, email, grade:"); scanf("%s%s%lf",id,email,&grade);
    insertLast(id,email,grade);
}
int main(){
    first = NULL; last=NULL;
    while(1) {
        char cmd[256];
        printf("Enter command: "); scanf("%s",cmd);
        if(strcmp(cmd,"Quit")==0) break;
        else if(strcmp(cmd,"Load")==0)   loadFile();
        else if(strcmp(cmd,"Print")==0)  printList();
        else if(strcmp(cmd,"Find")==0)   findProfile();
        else if(strcmp(cmd,"Insert")==0) insert();
        else if(strcmp(cmd,"Remove")==0) removeProfile();
        else if(strcmp(cmd,"Store")==0)  storeFile();
    }
    FREE_MEM();
}
```

41

Exercise 3: Profile management problem

- Change the grade of student with given studentID:
  - Ask studentID
  - Ask the new grade
  - Update the new grade for student with given studentID
- Count the number of students in the list passing the Math course (grade to pass the course: >= 5.0)

42

Contents

2. Stack

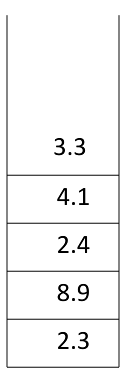
1. Implementation of stack using the linked list
2. An application of stack

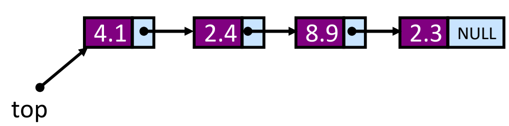
2. Stack

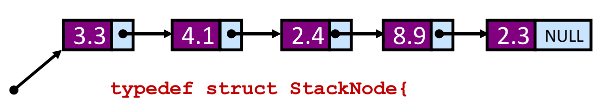
1. **Implementation of stack using the linked list**
2. Some applications of stack

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





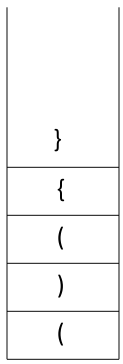


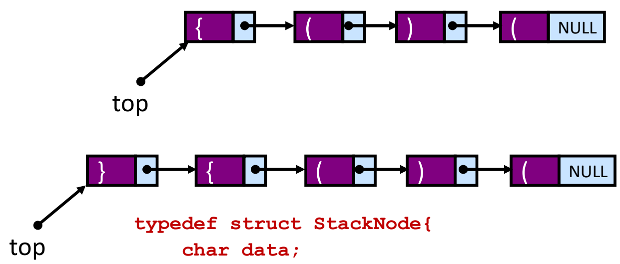
top

```
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 int(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

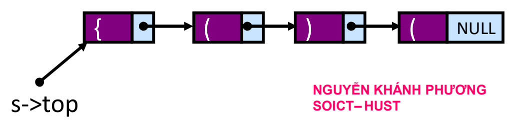
Need to do the following steps:

(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

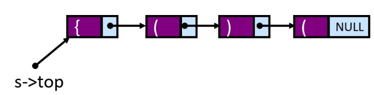
```
int push(Stack *s, char new_data) {
  StackNode *node;
  node = (StackNode *)malloc(sizeof(StackNode)); //(1)
  if (node == NULL) {// overflow: out of memory
    printf("Out of memory");return 1;
  }
  node->data = new_data;    //(1)
  node->next = s->top;     //(2)
  s->top = node;           //(3)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

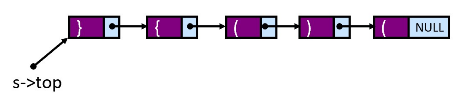


NGUYỄN KHÁNH PHƯƠNG
SOICT–HUST

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

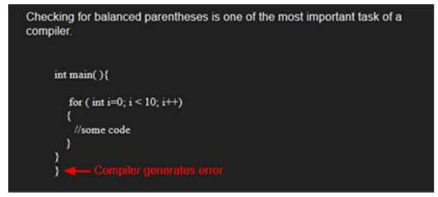
1. Implementation of stack using the linked list
2. **An application of stack:** Parentheses Matching

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"

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}

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

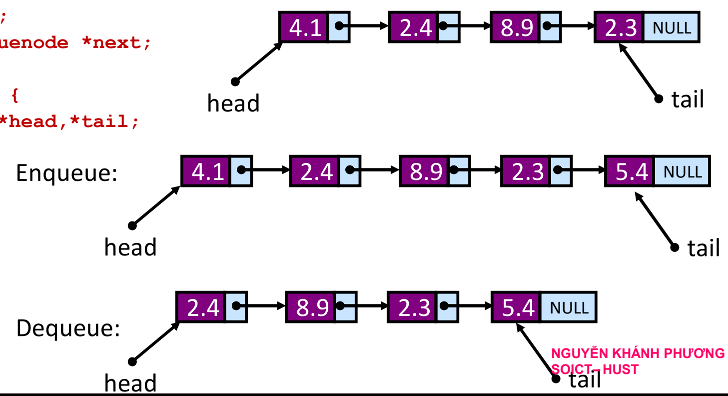
3. Queue

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

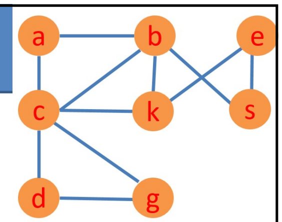
```
typedef struct Queuenode{
    float item;
    struct Queuenode *next;
}Queuenode;
typedef struct {
    Queuenode *head,*tail;
}Queue;
```



3. Queue

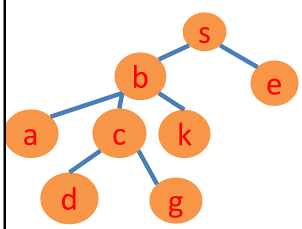
1. Implementation of queue using the linked list
2. **An application of queue**

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

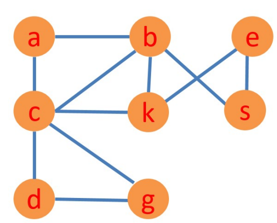
BFS creates a BFS tree containing s as the root and all vertices that is reachable from s

Adjacency list of s

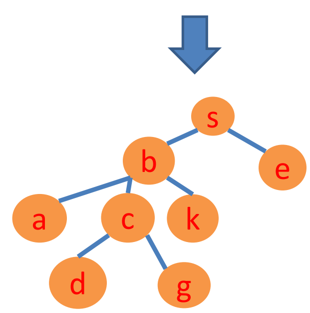
- From s: can go to b and e. Visit them and insert them into queue: Q = {b, e}
- Dequeue (Q): remove b out of Q, then Q = {e}
  - From b: can go to a, c, k, s. But s was visited, so we visit only a, c, k; and insert them into queue: Q = {e, a, c, k}
- Dequeue(Q): remove e out of Q, then Q = {a, c, k}
  - 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);
```

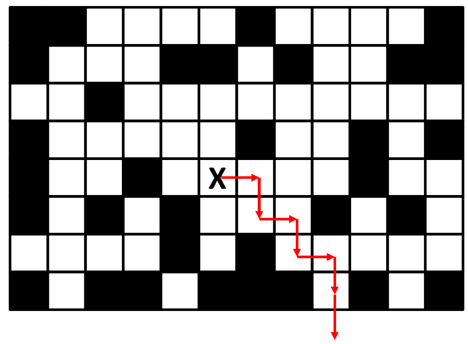


G=(V, E)



MAZE problem

- A Maze is represented by a 0-1 matrix $maze_{NxM}$ 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.

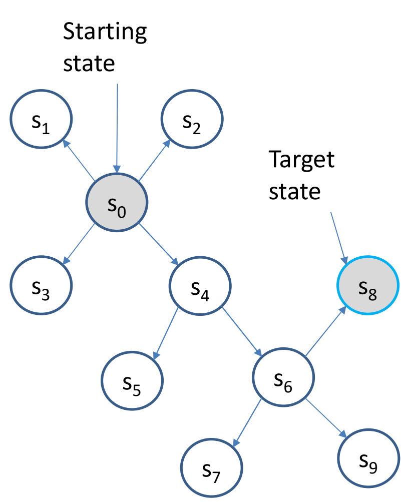


Escape the Maze after 7 steps

65

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 neighboring 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

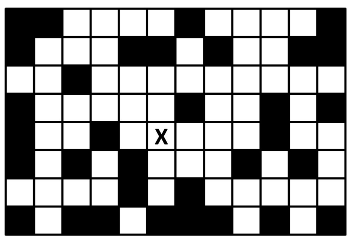
MAZE problem

```
#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");
```

NGUYỄN KHÁNH PHƯƠNG
SOICT – HUST